Procedure Graphs and Computer Optimizations

A thesis

submitted to

The Department of Computer Science

The Chinese University of Hong Kong

in partial fulfillment of the requirements

for the degree of

Master of Philosophy

by

Ho Kei Shiu Edward

June, 1992

ACKNOWLEDGEMENT

May I take this chance to express my deepest gratitude to my supervisor, Prof. Tien Chi Chen, for his patience and kind supervision as well as the invaluable opinions given in the past two years of my studies. And, with my hearty thanks, I would like to dedicate this piece of work to my family.

ABSTRACT

As hardware technology advances, the improvement in computer design seems to lag far behind. The weaknesses of prevailing architecture design philosophies soon become apparent. With the provision of a single program counter only, one is forced to examine the instruction stream sequentially, implying a strict total ordering in instruction executions. The fact that operations (or instructions) become unnecessarily ordered is the most crucial source of performance inefficiency.

To overcome this real "von Neumann bottleneck", we have to explore the causality relationships between instructions and discard any unnecessary execution constraint. By changing the original total ordering of operations into partial ordering, independent events are allowed to occur simultaneously. The degree of overlapped or parallel execution will be promoted as a result.

One may be familiar with the simple tagged architecture of the IBM 360/91 [Tomasulo67] and the optimizations done at the nodes of an interconnection network involving the Fetch-and-Add instructions [Chen91]. Promising results have been achieved. Behind these successful optimizing strategies is a set of rules. The underlying philosophy is that performance improvement can be realized via :

- redirecting data transfers
- deleting "unnecessary" arithmetic operations as well as "dummy" data transfers, and
- expediting all events as early as possible (so that the maximum parallelism can be exploited)

Unfortunately, for years, the subject has not been formally studied or even mentioned, and its potential in achieving optimization has not been fully developed and utilized, until T. C. Chen proposed the Procedure Graph Theory in his two papers [Chen&King89] and [Chen91]. Directed graphs are used for describing computer

operations and algorithms, with nodes representing storage locations or arithmetic operators, and directed arcs manifesting data transfers between them. The most important innovation is that pseudo-time labels on arcs are used to encode the global precedence relationship governing the correct executions of all operations and data transfers. Optimization of algorithm is achieved via transforming among equivalent procedure graphs representing the same computation. The fact that simple transformation rules can lead to very effective optimizations has motivated our current study.

Our discussions stem from the success of the procedure graph theory as a tool for precedence analysis. We are aiming at two major aspects of procedure graphs - theory and application. On the one hand, we hope that a complete mathematical model can be formulated with the concepts and theories of procedure graphs formalized. Possible extensions to the basic theory will also be explored.

Also, attempts will also be made to apply the theory, both as a general tool for modeling computer operations and an architectural design strategy for computer optimization. Our main focus will be on the implementation of transformation rules at the hardware level. We hope that through our study, weaknesses of prevailing computer architecture design philosophies can be uncovered. The insights provided may lead to a more efficient computation model which can realize the full benefits of self-optimizing computers. Our effort has given rise to a multiple-tagged architecture/representation, which is believed to be the true image of a procedure graph at the hardware level. Simulation results reveal that hardware level optimization alone suffices to achieve significant performance speedup even without the help of software techniques.

TABLE OF CONTENTS

Chapter 1 Introduction

1.1 Initial Motivation

For years, the classical execution model of computers has dominated the minds of the practitioners. But as hardware technology advances, the improvement in computer design seems to lag far behind. The weaknesses of prevailing architecture design philosophies soon become apparent.

In a classical uniprocessing machine, with the provision of a single program counter, one is forced to examine the instruction stream sequentially. The fact that operations (or instructions) become unnecessarily ordered is the most crucial source of performance inefficiency. We consider it (instead of the memory) as the real von Neumann bottleneck[1].

To tackle the problem, duplicating the hardware computing resources is the initial, easy step only. More importantly, we have to explore the causality relationships between instructions and discard any unnecessary execution ordering and constraint. By allowing independent events to occur simultaneously, the degree of overlapped or parallel execution can be promoted. While a total ordering of operations is implied by sequential programming languages and the von Neumann computation model, a partial ordering would be enough.

One may be familiar with the simple tagged architecture of the IBM 360/91 [Tomasulo67] and the optimizations done at the nodes of an interconnection network involving the Fetch-and-Add instructions [Chen91]. Promising results have been achieved. Behind these successful optimizing strategies is a set of rules. Their underlying philosophy is that performance improvement can be realized via :

[1] Many authors refer the memory system as the von Neumann bottleneck instead [Stone87]. They criticise that the provision of a single pair of Memory Address Register (MAR) and Memory Data Register (MDR) in the original von Neumann model limits the number of simultaneous memory accesses to 1 only.

- redirecting data transfers
- deleting "unnecessary" arithmetic operations as well as "dummy" data transfers, and
- expediting all events as early as possible (so that the maximum parallelism can be exploited)

Unfortunately, for years, the subject has not been formally studied or even mentioned, and its potential in achieving optimization has not been fully developed and utilized, until T. C. Chen proposed the procedure graph theory in his two papers [Chen&King89] and [Chen91]. Directed graphs are used for describing computer operations and algorithms, with nodes representing storage locations or arithmetic operators and directed arcs manifesting data transfers between them. The most important innovation is that pseudo-time labels on arcs are used to encode the global precedence relationship governing the correct executions of all operations and data transfers. Optimization of algorithm is achieved via transforming among equivalent procedure graphs representing the same computation. The fact that simple transformation rules can lead to very effective optimizations has motivated our current study.

1.2 Objectives Of Our Study

Our study aims at two major aspects of procedure graphs - theory and application. On the one hand, we will try to formalize the concepts and theories of procedure graphs. In particular, we are aiming at identifying more transformation rules. Possible extensions to the basic theory will also be explored. We hope that a complete mathematical model can be formulated.

Also, attempts will also be made to apply the theory, both as a general tool for modeling computer operations and a design strategy for computer optimization. Although we do not rule out the possibility and benefits of software optimizations, our main focus will be on the implementation of transformation rules at the hardware level. On the one hand, we

think that software optimizing techniques will complicate the design of compilers and lengthen the compiling time. More importantly, there are events not anticipated completely by compilers. We think that there exists a simple and effective hardware implementation of procedure graph optimization and we are confident that it alone suffices to achieve significant performance speedup. As a final comment, we hope that through our study, weaknesses of prevailing computer architecture design philosophies can be uncovered. The insights provided may lead to a more efficient computation model which can realize the full benefits of self-optimizing computers. Simulations will be done to justify our design decisions.

1.3 Outline of the Thesis

To begin with, chapter 2 will review the basic elements of the procedure graph theory originally discussed in [Chen&King89] and [Chen91]. Some formalizations are attempted, e.g. the classifications of nodes and arcs. The true meanings of the pseudo-time labels will be explored and a new formalism - the multi-level pseudo-time labels will be presented. The various dependencies limiting maximum parallelism or performance are considered (see [Hennessy&Patterson90], [Padua&Wolfe86] and [Stone87]) and their respective representations using procedure graphs will be presented. Attention is drawn to the use of procedure graphs for encoding causality relationships and the overriding of unnecessary ordering in instruction processing via equivalent graph transformations. At the end, we shall describe our simulation program for studying procedure graph transformations.

Based on the success of the original procedure graph theory, in chapter 3, we will try to extend it by introducing two new constructs - the T-Operator and the F-Operator. Our objective is to increase the modeling power and efficiency of the theory. Then, vector forwarding will be considered. The study does reveal some interesting properties concerning the definition and uses of algorithmic pseudo-time labels.

In the next three chapters, we turn to consider applying the procedure graph theory. Our main focus will be on hardware level optimization by dynamic re-scheduling. In search of an effective and efficient representation scheme, several approaches are compared and evaluated in chapter 4. Among them, the backward-pointer scheme is favored. A special implementation based on hardware tags will be studied, which was first adopted in the IBM 360/91 [Tomasulo67]. Equivalent graph transformations are achieved via simple manipulations of tags done in real-time.

Significant enhancements are incorporated into this original model, giving rise to the T-Architecture. Initially evolved as a superscalar design [Johnson91], the T-Architecture represents an integration of various optimizing strategies - procedure graph transformations, memory data forwarding, speculative execution[2], etc. In chapter 5. we will consider these features in detail. Simulation results show that a performance level of as high as 96% of the maximum efficiency can be attained without the help of software optimizing techniques. The success of the T-Architecture has exemplified the benefits of backward-pointer representation schemes.

In spite of this, the T-Architecture does contain certain weaknesses. First, with backward pointers, only upstream arc-information can be directly obtained via traversing the singly-threaded list in the backward direction. This asymmetry has inhibited, to some degree, the applications of certain graph transformations (e.g. the Parallel-to-Serial Transformations). On the other hand, the fact that only a single tag can be associated with each node has constrained us to do real-time optimization only. The restriction in the scope and lookahead capability has limited the effectiveness of the results obtained.

---

[2] Speculative Execution (and such related concepts as the implementation of precise interrupts and the repair of branch fault) have been mentioned or studied by various authors before (see [Smith et al.90], [Smith&Pleszkun88] and [Hwu&Patt87]). A special design using backward pointers is adopted in the T-Architecture.

In view of these, another design, the S-Prototype, is proposed in chapter 6. As an attempt to achieve predictive optimization, multiple tags can now be associated with each node which are centralized in a "scoreboard-like" Multitag Pool. Basically, a doubly-threaded list representation is adopted, allowing more efficient manipulations of procedure graphs. We believe that the multiple-tag representation scheme is the true image of a procedure graph at the hardware level. This way, we are in fact working towards a hardware implementation of simple compiler optimization techniques.

In chapter 7 and chapter 8, we shall approach the problem from a different viewpoint. Two related concepts will be explored. First in chapter 7, we shall describe our attempt to simulate procedure graph transformations using graph grammar (see [Ehrig79] and [Ehrig87]). Though the study turns out to be incomplete at last, it does provide insights concerning certain interesting characteristics of procedure graph transformations. And analogies of various graph grammar concepts can be identified in the procedure graph theory (e.g. parallelism and concurrency). In chapter 8, the Petri net theory is studied [Peterson81]. The main focus will be on program analysis and computer operations optimization. Through this study, we hope that we can evaluate the expressive power and economy of procedure graph theory and more importantly, the insufficiencies uncovered, if any, can help to reveal possible extensions to the basic procedure graph theory.

Finally in chapter 9, we will summarize what we have done in the past two years. As a conclusion to this thesis, future research directions are outlined.

Chapter 2 Basics of the Procedure Graph Theory

2.1 Introducing Procedure Graph Theory

Ever since its birth in 1736 when L. Euler posed and solved the Königsberg bridge problem in his famous paper [Deo74], Graph Theory has been extensively applied to various fields and disciplines including mathematics, engineering sciences and social sciences, etc, successfully solving a wide range of problems.

Computer science is no exception. We assume that the reader is familiar with the use of directed graphs for describing the precedence of computer operations (process graphs or precedence graphs), as depicted by the example in figure 2.1 (where A, B and C denote processes to execute).

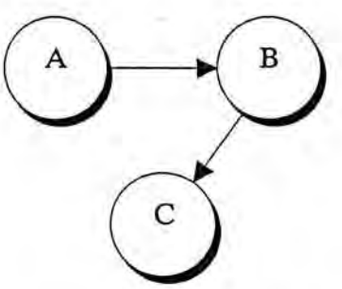


Figure 2.1. A Process Graph

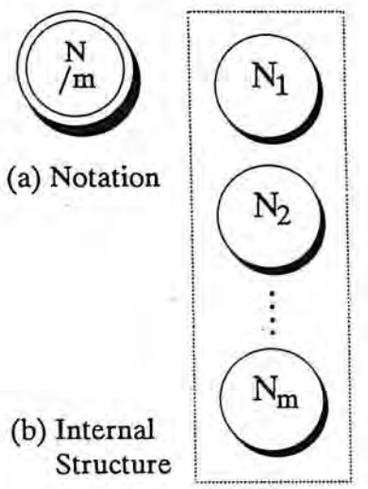
T. C. Chen put this success one step further by introducing the Procedure Graph Theory (see [Chen&King89] and [Chen91]). Directed graphs are used for describing computer operations and algorithms, with emphasis on the flow of data. Nodes are used to represent storage locations and arithmetic operators (or operations) while data transfers are manifested by directed arcs. The most important innovation is that pseudo-time labels are used on arcs to encode the precedence or causality relationship governing the correct order of data transfers and operations.

In this chapter, we shall present the basic concepts of the procedure graph theory. Most of the ideas were originated by T. C. Chen [Chen91]. New contribution will be specially highlighted when appears.

2.1.1 Nodes, Arcs and Pseudo-time Labels

In a procedure graph, nodes are used to denote storage locations or arithmetic operators. Precisely, storage nodes can be further divided into two types - Simple Nodes and Complex Nodes. As its name suggests, a simple node represents a single storage location such as a scalar register or an addressable memory cell. On the other hand, an array of storage locations can be manifested by a complex node. Intuitively, a complex node can be perceived as containing more than one simple node. The notion of a complex node is useful for describing a vector register or a memory bank.

Figure 2.2.
Complex Storage Node



A new notation is introduced here. Figure 2.2(a) depicts a complex node N consisting of m simple nodes as indicated by "/m". Its internal structure is illustrated in figure 2.2(b), showing the individual components N_1, N_2, ..., N_m.

Arithmetic operators are classified in much the same way. According to the original procedure graph theory, a simple arithmetic node produces an output based on the set of valid input operands fed to it via the entry arcs. This concept can be generalized, allowing the representation of another procedure graph by a single arithmetic node. This gives rise to a complex arithmetic node. In such a case, entry arcs and outgoing arcs of this complex arithmetic node describes its "interface" to the remaining global graph. When computer operations are simulated, a simple arithmetic node will then correspond to a single functional unit while a sub-algorithm is abstracted by a complex arithmetic node.

By definition, a data transfer (event) from a source node A to a sink node B occurring at time $f(T)$ will be represented by a directed arc leading from A to B

annotated by a pseudo-time label T. Intuitively, a pseudo-time label can be perceived as a number specifying the order of executing the corresponding data transfer. As a general rule, for two arcs α and β annotated with the pseudo-time label T1 and T2 respectively such that T1<T2, the data transfer corresponding to the arc α will happen before that of β. When T1=T2, their precedence is immaterial and they can be invoked at the same time. "Parallelism" of this kind serves as a major source of optimization. To avoid inconsistencies, the pseudo-time labels of two data transfer arcs leading to the same sink node N cannot be equal, unless N is an arithmetic node or a complex storage node.

In accordance to our earlier discussions for nodes, we also distinguish between simple and complex data transfer arcs. When both the source and sink are simple nodes[1], the corresponding data transfer will be described by a simple arc, as exemplified by figure 2.3, where Ri and Rk represent simple storage or arithmetic nodes such as scalar registers.

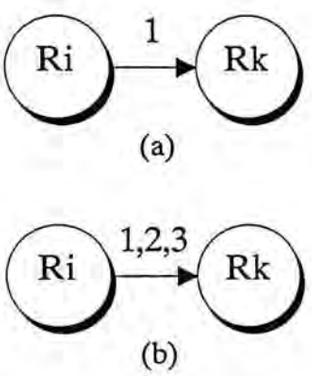


Figure 2.3. Simple arcs

Otherwise, a complex arc is implied, which in general will be annotated by multiple (or a list of) pseudo-time labels $\{t_i\}$ instead of just one[2], each corresponds to an individual data transfer event. As shown in figure 2.4, new notations have been introduced for complex arcs. A "bubble" is added to the starting point (respectively the

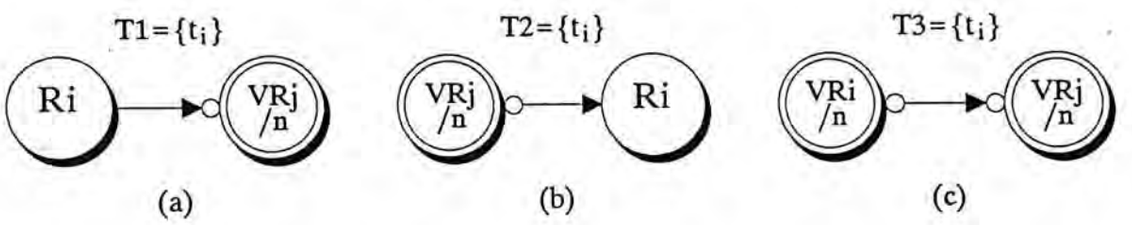
[1] The situation becomes a little bit complicated when complex arithmetic nodes are involved. Let's illustrate it with an example. Given an arc N1→N2 where N1 is a simple node and another subgraph G' is abstracted by N2. We substitute N2 by G'. Arcs previously with N2 as the source or the sink will now be connected to some node in G'. Without exception, there should exist a node N in G' which the original arc N1→N2 will now be incident on. Depending on the type of N (a simple storage or arithmetic node, or a complex storage node), we can classify N1→N (or N1→N2) as a simple arc or complex arc.

[2] In general, an algorithm will be used which gives rise to a list of pseudo-time labels upon enumeration. See later discussions for details.

terminating point) of a data transfer arc if its source node (respectively the sink node) represents a complex storage node.

Multiple time labels can also be used for a simple arc, should a repeated data transfer over time from a simple node to another be described. However, these events involve the same source and sink and they are in fact mutually independent.

Figure 2.4. Complex arcs



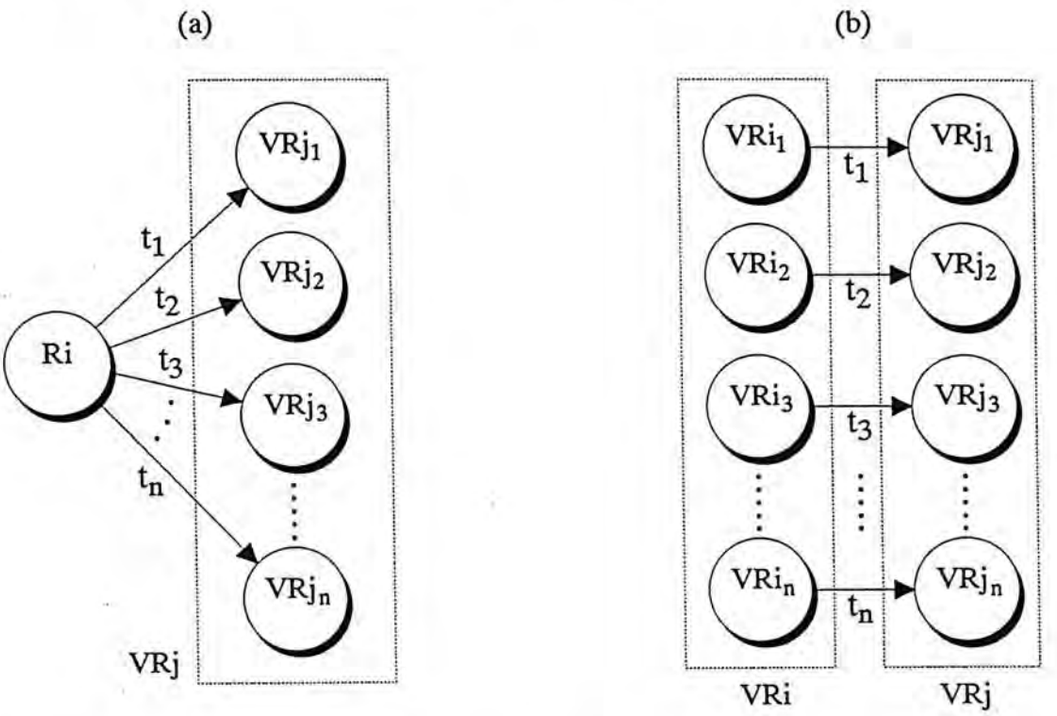
Key :
Ri : Simple storage or arithmetic nodes, e.g. scalar registers
VRi, VRj : Complex storage nodes, e.g. vector registers

On the one hand, a complex arc can represent multiple or repeated data transfers over time from a simple node to a complex node or vice versa, as illustrated by figures 2.4(a) and 2.4(b). Drawn in figure 2.5(a) is an example showing the initialization of a vector (corresponding to figure 2.4a). At each time t_i (where i=1,2,...,n), a different component VRj_i of the vector register VRj is involved. Depending on the actual implementation of VRj, these data transfer events may be invoked serially or in parallel. In the latter case, a list of equal time labels will be associated with the complex arc. Without ambiguity, we may adopt a simple scalar pseudo-time label instead, such as T1=1, effectively (and in fact sufficiently) expressing that individual components of the VRj are accessed simultaneously. Similar argument applies for T3 also.

On the other hand, when both the source and the sink are complex nodes, a vector data transfer between the corresponding storage locations is implied, as depicted

in figures 2.4(c) and 2.5(b). Again, the individual (scalar) data transfer events can be executed serially or in parallel.

Figure 2.5. The space-time relationship as described by a complex arc



Key :
Ri : Simple storage or arithmetic node, e.g. scalar register
VRi, VRj : Complex storage nodes, e.g. vector registers

Whatever interpretation is desired, we can see that a space-time relationship is encoded by a complex arc, whereas only the concept of time is involved for a simple arc. As a final comment, in case only specific element VRj_i of a complex node VRj is of interest instead of the whole VRj, a simple node will be dedicated for it (exclusively) which is labelled by "VRj_i". The corresponding data transfer event(s) will then be represented by a simple arc, similar to the situations shown in figure 2.3.

As a final comment, the introduction of complex arcs may suggest the need for a new notation, e.g. a different type of arrow-head, to distinguish them from simple arcs in case they should appear together. In succeeding discussions, unless stated explicitly otherwise, we refer to simple arcs only.

Although arc-labels are common since the first definition of graphs, they used to represent only weights, lengths, or costs, etc [Deo75]. The idea of using arc-labels for encoding an element of time is non-trivial. As pointer out by T. C. Chen [Chen91], two advantages of pseudo-time labels on arcs (over labelling nodes) are obvious. First, the use of pseudo-time labels is direct and intuitively more appealing. In addition, there is more room to specify multiple time labels on arcs than on nodes.

The usefulness of the pseudo-time labels can be made explicit by examining the example in figure 2.6(a) where there is a feedback from process C to process A. Using the classical (non-precedence) graph theory, we just don't know how to resolve the cycle (say by a topological sort). But in figure 2.6(b), the desired or correct order of operations is governed explicitly by the pseudo-time labels, avoiding the ambiguity in figure 2.6(a) completely.

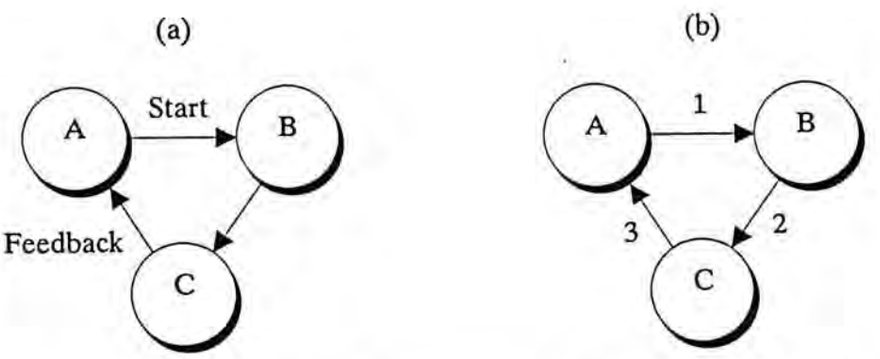


Figure 2.6. Demonstrating the usefulness of the pseudo-time labels

As a final comment, procedure graph theory gives rise to a new computation model. To solve a problem, we think of an algorithm which is represented by a global procedure graph (possibly a complex graph) with pseudo time labels. To optimize operations, we extract/map a subgraph of it, identify an equivalent graph, and then perform the transformation. The resulting subgraph is stitched back into the original graph. Having preserved the causality of operations and data transfers, the transformed graph will represent the same computation as before.

In fact, we are looking for equivalences of sub-algorithms. Every time we focus on a subset of the total computation (which is not constrained to be the very beginning of the algorithm). By replacing it by a more efficient equivalent graph, the overall performance can be enhanced. The fact that the computation performed will not be affected is guaranteed by the rule of associativity that holds among the different sub-algorithms of a program.

2.2 Examples

Let's consider the procedure graph depicted in figure 2.7(a) representing the following two instructions :

```
LD    F0,A[3]
ADD   F2,F0
```

With reference to figure 2.7(a), the desired interpretation is :

At time 1 The content of the memory location A[3] is being loaded into the register F0. At the same time, the value of another register F2 is read as the left operand of the ADD operator.

At time 2 The content of F0 (previously delivered from A[3]) is "relayed" to the ADD operator as its left operand. Having received its both operands, the ADD operation is fired.

At time 3 The output/result of the ADD operation F2+F0 is written back to the sink register F2.

As shown, a partial ordering has been imposed explicitly by the pseudo-time labels which governs the correct and necessary causality relationships that should exist among the various operations as dictated by the original instruction sequence.

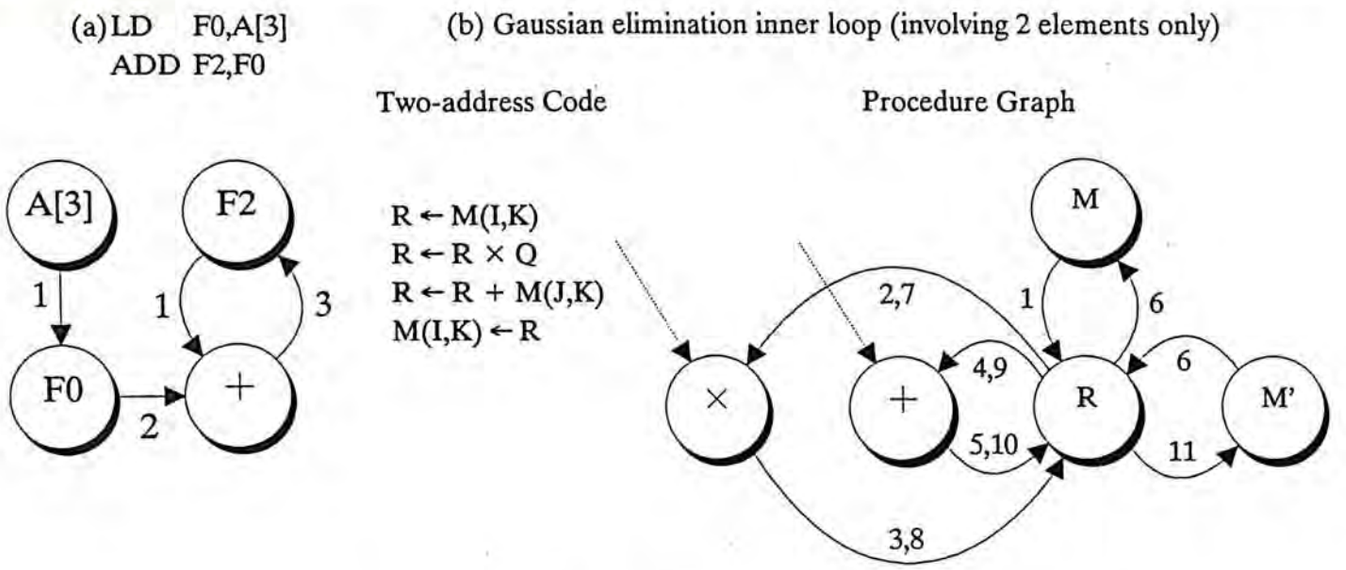


Figure 2.7. Examples illustrating the use of procedure graphs

As another illustration, the Gaussian elimination inner loop (involving two elements) is considered. Figure 2.7(b) exhibits the corresponding procedure graph. Multiple pseudo-time labels on arc allow the specification of repeated data transfers along the same path over time (events happening at different time intervals). Just as a note, we can see that the assignment of the pseudo-time labels agrees with the original instruction execution sequence, resulting in a strict sequential ordering of operations. Doubtless, some of these causality constraints are in fact unnecessary, which when removed, may realize significant speedup in processing.

2.3 Exploring the Meanings of the Pseudo-time Labels

The use of pseudo-time labels sounds simple enough. Yet, the underlying assumptions should not be overlooked. To begin with, we should emphasize that the pseudo-time label needs not be instantiated with the actual cycle time, although they should be closely related. Hence, for example, the use of the two pseudo-time labels T1=1 and T2=2 does not necessarily imply that their respective flows of data should happen at exactly one cycle apart. All we can deduce is that the first one is invoked before the second and nothing more.

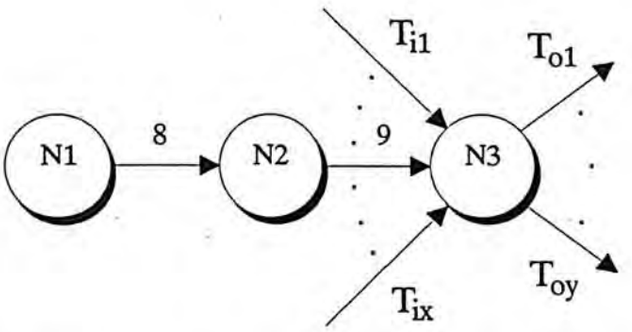


Figure 2.8. Pseudo-time label as the most appropriate value in a range

Next we proceed to examine the true meaning represented by a pseudo-time label. Consider the situation exhibited in figure 2.8. As shown, the node N3 has a set of output arcs $\{O_1, ..., O_y\}$ labelled by the pseudo-time labels $\{T_{o1}, ..., T_{oy}\}$ respectively and a set of input arcs $\{I_1, ..., I_x\}$ annotated by $\{T_{i1}, ..., T_{ix}\}$, such that the flows of data depicted by those output arcs occur after the transfer of data from N2 to N3 which in turn happens later than the flow-in of data described by the set of input arcs $\{I_1, ..., I_x\}$. Let :

$T1 = \min \{ T_{os} \}$	where $s = 1, 2, ..., y$	and
$T2 = \max \{ T_{it} \}$	where $t = 1, 2, ..., x$	

The entire set of leaving arcs $\{O_1, ..., O_y\}$ is guaranteed to carry the same data item transferred into N3 by the arc labelled "9", if

$T2 < 9$ and $T1 > 9$

and there exists no entry arc into node N3 with pseudo-time label T such that $9 \leq T < T1$. In fact, we can assign any positive numeric label L on the edge leading from N2 to N3 such that

T2 < L < T1

Please be noted that the argument above applies for N3 is a storage node (register or memory). If it is an arithmetic node (representing, say, an addition operation), we will have $T2 \leq L < T1$ instead. Now we can give a more appropriate interpretation of pseudo-time labels. The label "L" used in the above discussion in fact denotes the most probable or desirable time label (that is, a typical value only), and in principle, it should be treated as a variable or representing a range of valid values. Analogously, we say

'After you have finished job1 and before you proceed to job3, complete job2.'

The use of an appropriate instance of a range is more suitable for our application than labelling nodes with precedence bounds, as it provides greater flexibility when multiple pseudo-time labels are to be represented. It is intuitively simpler and considerable convenience is offered when we have to deal with graph transformations. Further, reassignment of labels present no problem at all.

A more general interpretation of pseudo-time labels should allow the representation of algorithmic time labels. For example, it may allow specifications like $i+1$, where $i=0,1,2,...$,etc. This flexibility is especially appealing when vector data are involved. Each new instance of i corresponds to the transfer of the next vector component. It may sound confusing, but a similar notation will be used for encoding repeated data transfers from the same scalar location. Both give rise to a list of pseudo-time labels on enumeration. We will come to this issue again in chapter 3 when we consider vector forwarding.

To conclude, in any procedure graph, a global order governing the flows of data can be specified by a set of pseudo-time labels forming a non-decreasing sequence of

numbers. The use of equal time labels implies that their corresponding flows of data can be overlapped, thus revealing chances of parallelism concerning non-interfering events.

2.4 Equivalence and Transformation

Prescribed execution sequence can be changed into another and still produces the set of results. The two sequences are said to be equivalent. Transformation can be done if causality can be preserved. Even if it is broken, it can be restored. More importantly, sometimes causality is actually irrelevant, and can be discarded at will. Thus any sequence of positive integers can be assigned to the data transfer arcs of a procedure graph, if the transformed graph achieves the same purpose intended. This is the fundamental rationale of graph transformations and pseudo-time labels renumbering.

2.4.1 Equivalence

Two procedure graphs G and G' are said to be equivalent, written as $G \Leftrightarrow G'$, if the computations represented by them produce the same set of results upon completion under identical inputs. From another point of view, they leave the same final contents in each storage location of interest.

2.4.2 Transmission Track and Causality Preservation

According to the original procedure graph theory [Chen91], the directed path $N_1 \rightarrow N_2 \rightarrow ... \rightarrow N_p \rightarrow N_{p+1}$ in figure 2.9 forms a track of length $p > 0$ if $T_1 < T_2 < \cdots < T_p$ and there exists no incident arc into any of the intermediate node N_k $(2 \leq k \leq p)$ with time label T satisfying $T_{k-1} \leq T < T_k$. N_1 and N_{p+1} are designated as the source and sink respectively. Given an input data α at N_1, we can obtain at N_{p+1} (after a time T_p) an output β. If none of the intermediate nodes N_2, N_3, ..., N_p is an arithmetic node, it is a transmission track. Otherwise, we have an arithmetic track instead.

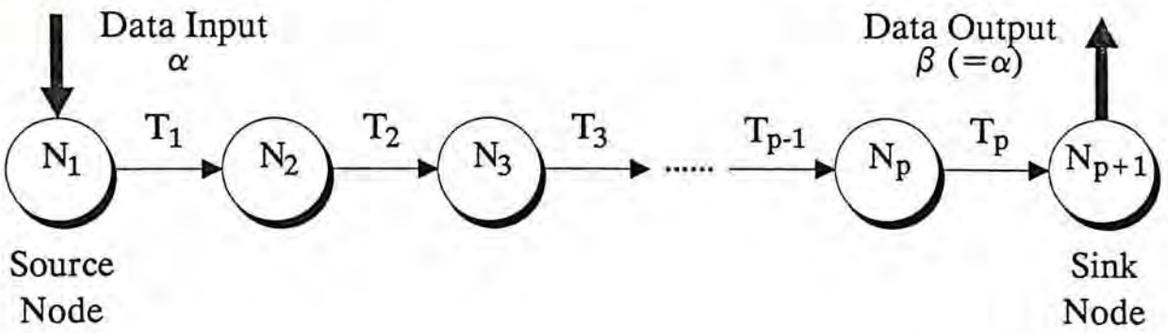


Figure 2.9. Causality relationship exhibited by transmission tracks ($T_1<T_2<T_3<\cdots<T_{p-1}<T_p$)

A procedure graph G can be perceived as constituted by a number of transmission and arithmetic tracks [Chen91]. Together, they represent an algorithm with its input requirements/nodes N_{input} specified by the set of source nodes N_{source}. In general, only a subset N_{output} of the total set of sink nodes N_{sink} is designated as the output nodes. Upon a set of input data $\{\alpha_i\}$, G produces a set of outputs $\{\beta_j\}$. No matter how G is transformed, the same $\{\alpha_i\}$ should give rise to the same $\{\beta_j\}$. This is the fundamental rationale of graphical equivalence.

As illustrated by figure 2.9, every transmission track possesses the useful property that if we put a data α at its source (N1), we will obtain β at a later time T_p from its sink such that $\beta=\alpha$. In this sense, the phenomenon exhibits a causality relationship, which in general does not hold for an arithmetic track because the input data α will be transformed when it travels along the track, giving rise $\beta=\beta(\alpha)$ but usually $\beta\neq\alpha$. It is this causality relationship that is of special interest to us and in our succeeding discussions, we shall concentrate on procedure graph transformations preserving this kind of causality.

2.4.3 Transformation

Equivalent graphs can be transformed among each other via causality-preserving transformation. Generally speaking, a single equivalence G⇔G' gives rise to two

possible transformations G⇒G' and G'⇒G. Many transformation rules have been mentioned by T. C. Chen [Chen91]. Among them, we are especially interested in the three classical internal forwarding rules - Serial-to-Parallel Transformation (SP), Parallel-to-Serial Transformation (PS) and Store-Store Cancellation (SSC).

2.4.3.1 Serial-to-Parallel Transformations (SP)

Consider the situation depicted in figure 2.10(a). Given a transmission track N1→N2→N3, the data delivered from N1 to N2 at time T1 will be transferred to N3 at a later time T2. By executing a Serial-to-Parallel Transformation, written as SP(N1 N2 N3) assuming pre-transformation arc sequence, an equivalent graph (implying an alternative computation) can be obtained, as shown in figure 2.10(b).

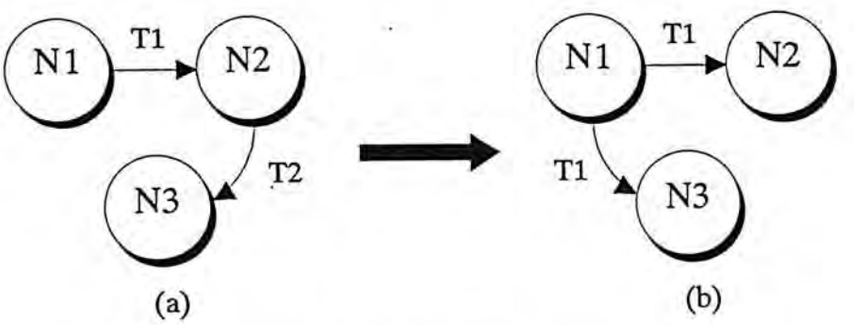


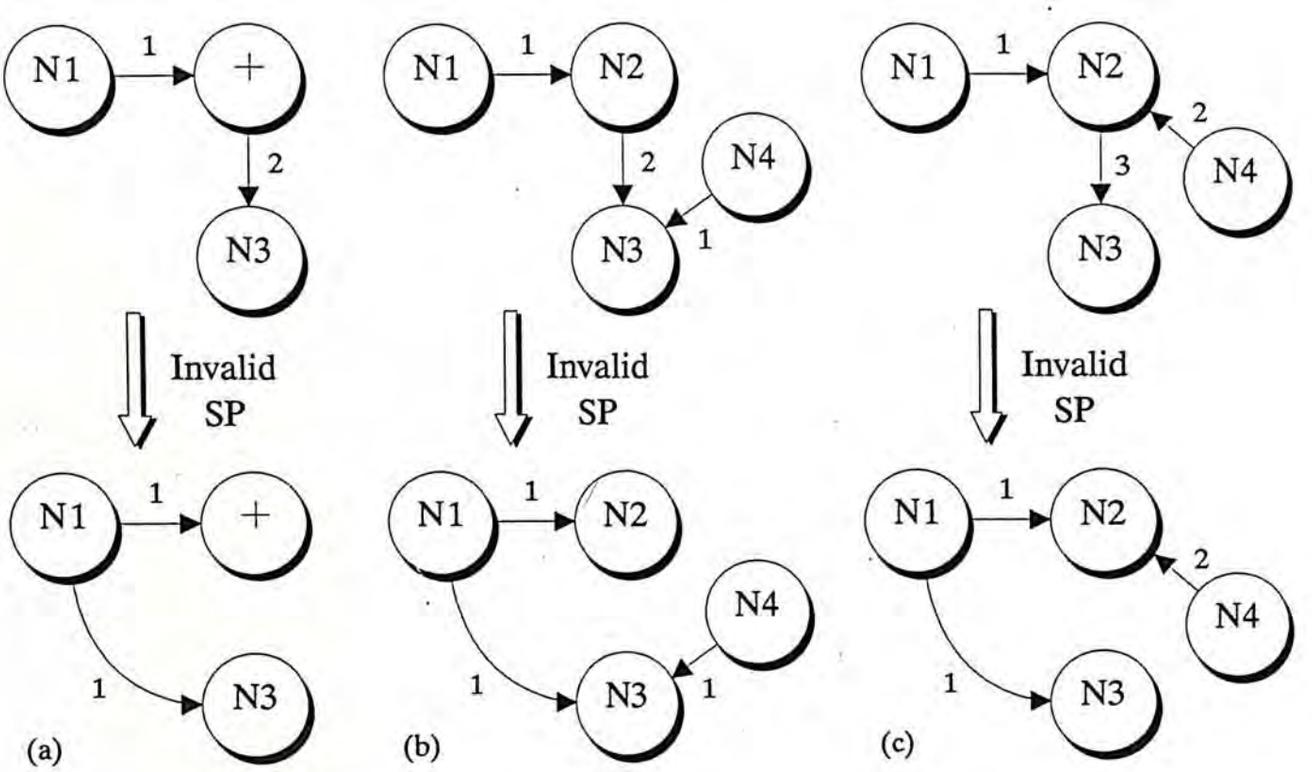
Figure 2.10. A Serial-to-Parallel Transformation (T1 < T2)

The assumption that N1→N2→N3 forms a transmission track on the one hand implies that N2 must be a storage node (although N1 and N3 can represent arithmetic nodes). On the other hand, it guarantees that there exists no entry arc into the node N2 with pseudo-time label T satisfying $T1 < T < T2$. To preserve causality, we have to assure further the absence of any incoming arc into N3 annotated with a time label T' such that $T1 \leq T' < T2$. Together, these three application constraints safeguard a legitimate Serial-to-Parallel Transformation.

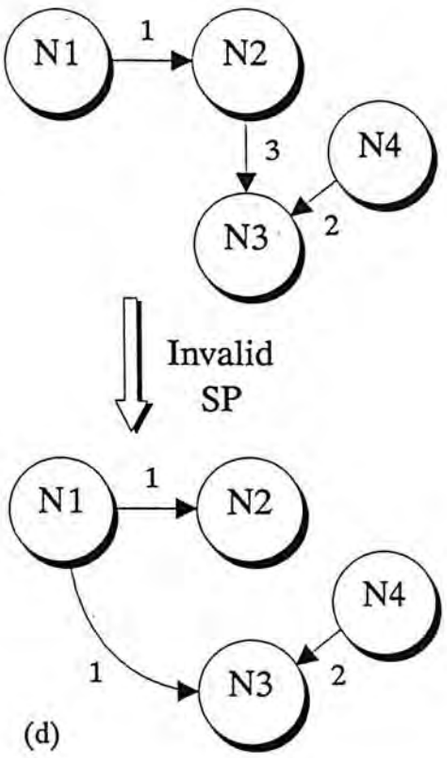
As its name suggests, the relayed data transfers (N1 to N2 and then to N3) are changed into parallel "broadcast" (N1 to both N2 and N3 simultaneously) via the SP Transformation. Two advantages are obvious. First, the total duration is shortened. Moreover, in case the node N2 is a memory location, one memory (read) access is saved.

Although the transformation rule sounds simple enough, the constraints of application should not be overlooked. Some of them are summarized in figure 2.11. To begin with, if N2 happens to be an arithmetic node such as the add operator in case (a), the data written to N3 is "transformed" instead of the original copy from N1. As a result, we cannot apply the SP or the final content of N3 will be incorrect.

Figure 2.11. Examples showing invalid applications of Serial-to-Parallel Transformations



On the other hand, the fact that we have two entry arcs leading to the same sink node N3 annotated with equal time label "1" results in an inconsistent situation which prohibits the SP in case (b) (an exception being the case when N3 is an arithmetic node). Case (c) and case (d) address similar problems. The ignorance of the information of surrounding arcs results in the violation of the original causality relationship, leaving incorrect final content in the node N3.



2.4.3.2 Parallel-to-Serial Transformations (PS)

Intuitively, a Parallel-to-Serial Transformation may be perceived as the reverse of the corresponding Serial-to-Parallel Transformation. With reference to figure 2.12(a), the "fanout" from N1 to N2 and N3 is changed into a serial transfer - first from N1 to N2 at time T1, and then to N3 at a later time T2, as manifested by the equivalent procedure graph in figure 2.12(b). For notation, we write PS(N1 N2,N1 N3), again assuming pre-transformation arc sequence.

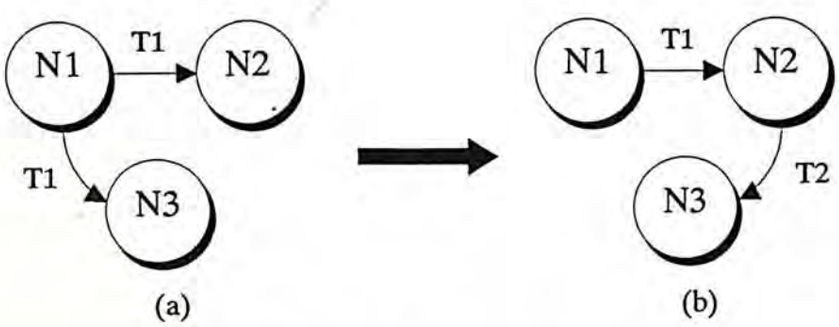


Figure 2.12. A Parallel-to-Serial Transformation (T1 < T2)

The PS saves us from one memory access if the source N1 is a memory location. At the same time, the "fanout" requirement at N1 is also relaxed. However, the total duration of the process is increased as a result.

Again, we consider the application constraints of PS. In figure 2.13, we have tailored examples for illustration. The interpretation for situations in case (a) and case (b) is straight-forward and analogous to our earlier discussions for SP. For case (c), the consequence that the data to be stored in N4 becomes undefined makes the PS invalid.

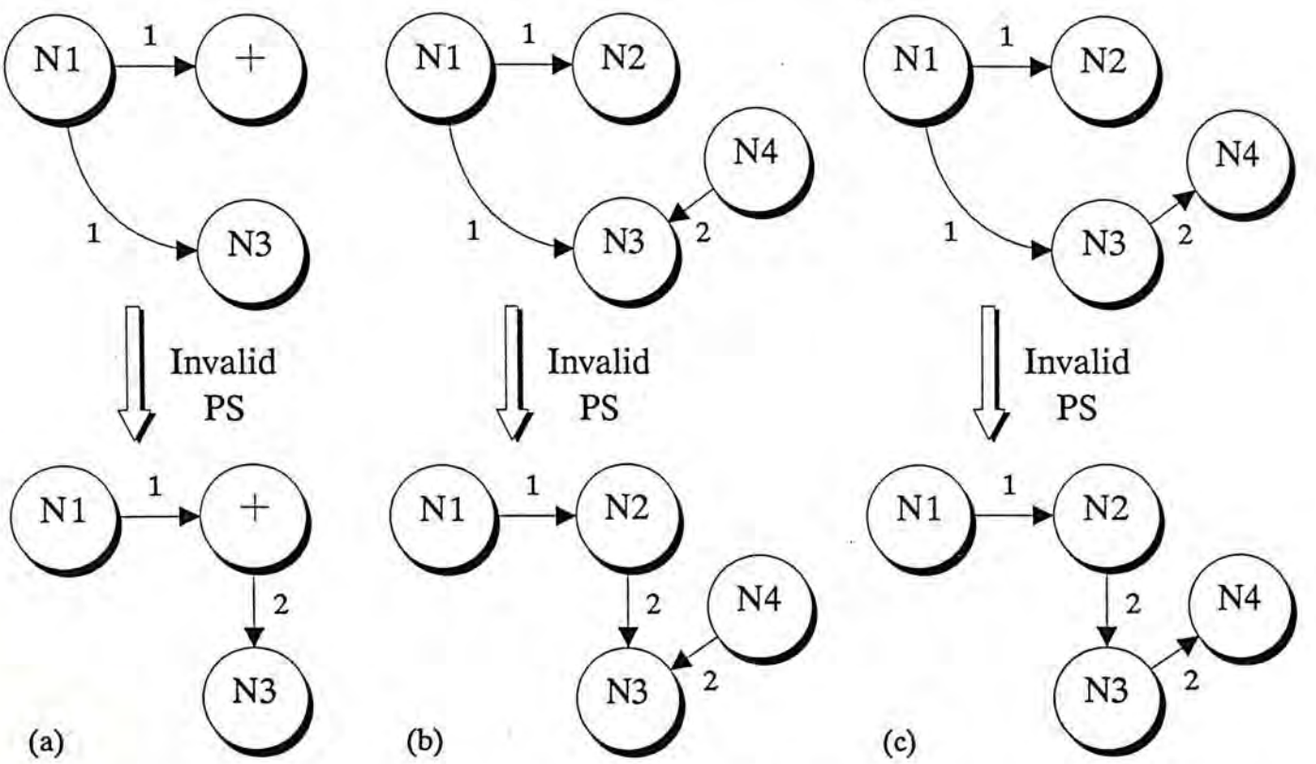


Figure 2.13. Examples showing invalid applications of Parallel-to-Serial Transformations

2.4.3.3 Store-Store Cancellations (SSC)

Suppose we have two data transfers to the same sink node labelled by T1 and T2 respectively such that $T1 < T2$, as exemplified in figure 2.14. If it happens that there exists no outgoing arc from N3 with time label T satisfying $T1 < T \leq T2$, we can

overwrite N1→N3 by N2→N3. Effectively, the Store-Store Cancellation SSC(N1 N3,N2 N3) deletes the dummy data transfer N1→N3.

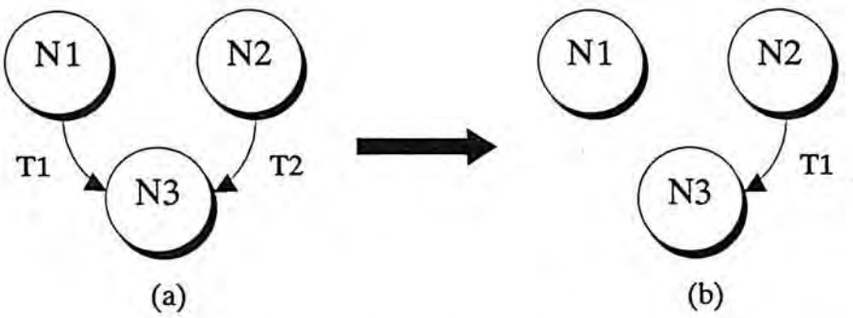


Figure 2.14. A Store-Store Cancellation (T1 < T2)

Before applying an SSC, we should make sure that there exists no incoming arc with time label T such that T1 < T < T2 or outgoing arc with time label T' such that T1 < T' ≤ T2, lest the situations in figure 2.15(a) and 2.15(b) will result.

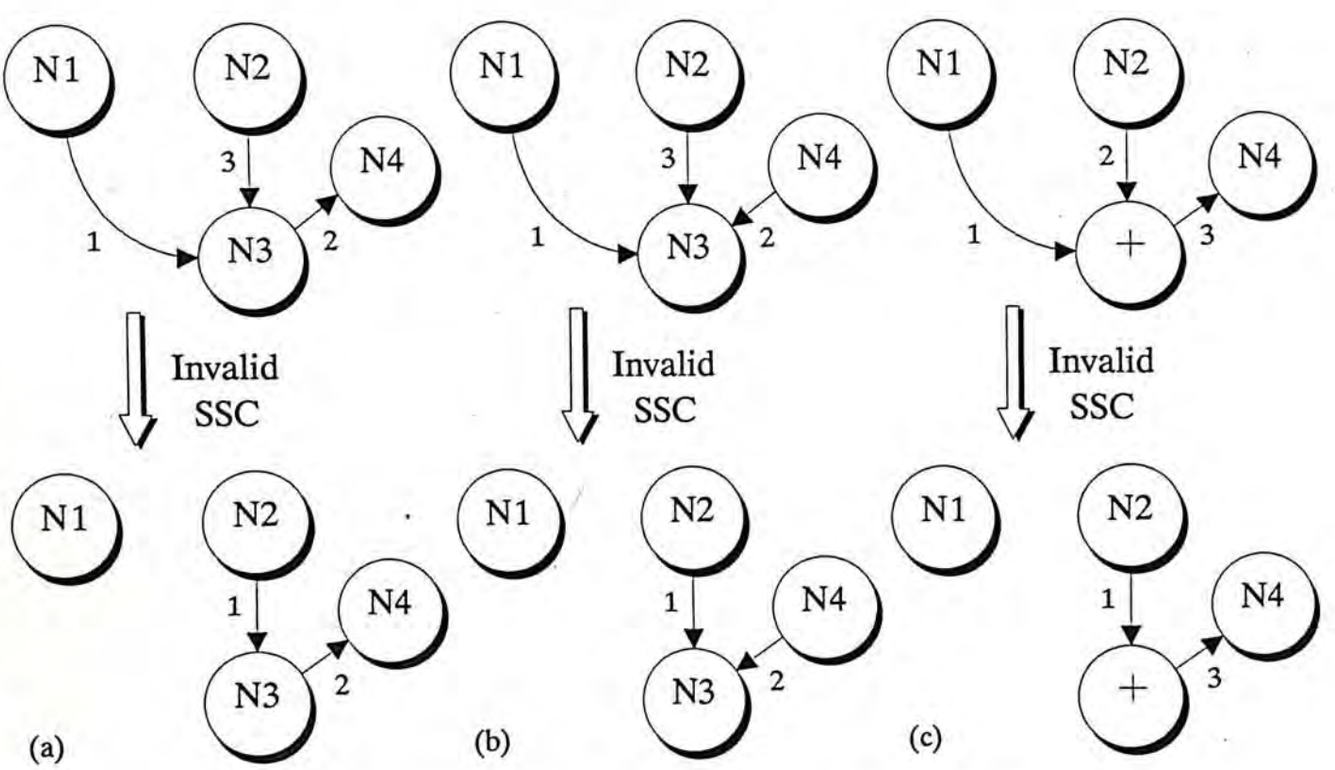


Figure 2.15. Examples showing invalid applications of Store-Store Cancellation

Moreover, in case N3 is an arithmetic node as illustrated in figure 2.15(c), we should check the arity of the operator to avoid deleting the supply of operands unexpectedly. In particular, the arc N2→+ should not overwrite the arc N1→+.

2.4.3.4 Normalization of Pseudo-time Labels

Sometimes two procedure graphs may look different. But the fact that they are inherently equivalent will soon become apparent upon normalization of the pseudo-time labels. This re-labelling of time labels on the one hand preserves the meaning/computation of a procedure graph, and at the same time expedites each data transfer as early as possible. Normalization of time labels helps proving graphical equivalence. More importantly, opportunities of parallel transfers are explored such that the total computation time can be shortened. Here we contribute a formal and systematic way to achieve these.

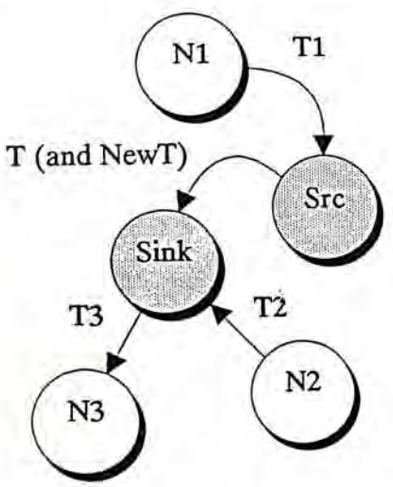


Figure 2.16. Normalization of pseudo-time labels

In general, normalization is done by accepting bounds to a pseudo-time label, and then selecting the smallest possible value. With reference to figure 2.16, suppose we want to normalize the time label T of the arc leading from the node Src to the node Sink. First, we have to define the followings :

T1 = max {T' such that ∃arc(N1,Time,Src) and T'∈Time and T'<T}
T2 = max {T' such that ∃arc(N2,Time,Sink) and T'∈Time and T'<T}
T3 = max {T' such that ∃arc(Sink,Time,N3) and T'∈Time and T'≥T}

Prolog-like notations have been adopted here. For example, "arc(N1,Time,Src)" denotes an arc leading from N1 to Src which is annotated by a list "Time" (possibly of one element only) of pseudo-time labels. The assertion "T'∈Time" will be true if T' is contained in the list of time labels Time.

The new time label "NewT" to replace T should be greater than T1 since it is the piece of data that the arc labelled T1 brings into Src is transferred to Sink at T (or NewT). In addition, NewT should be larger than T2 also. The rationale is that if NewT is smaller than T2, the data output to N3 will be wrong and the final content of Sink will be changed also. Finally, the transfer from Src to Sink should not interfere with the previous output from Sink. Thus we have NewT ≤ T3.

To summarize, any value satisfying the above three criteria is acceptable[3]. But to expedite data transfer as early as possible, we should have :

$$\boxed{\text{NewT} = \max\{\text{T1},\text{T2},(\text{T3}-1)\} + 1}$$

Every time when one or more of T1, T2 and T3 change(s), the value of NewT has to be recalculated. As a final comment, if there exists no entry arc into the node Src with time label smaller than T, then we will set T1 to zero. Similar argument applies to the case of T2 and T3.

2.4.3.5 Boundary Conditions and Multi-level Pseudo-time Labels

As revealed by the discussions in the previous section, a direct consequence of normalization is that the pseudo-time labels are constrained to have integer values only.

---

[3] This is true for each pseudo-time label in any procedure graph. Thus we can see that there is a set of inequalities governing the pseudo-time labels. By enumerating all combinations of legitimate values of these time labels, a set of equivalent graphs can be obtained. This agrees with our earlier argument that a pseudo-time label is in fact the most probable value of a range only.

While considerable convenience is provided in graph transformation and time label renumbering, the use of integers does imply certain undesirable side-effects.

Consider the situation depicted in figure 2.17. The node N2 in G actually represents a subgraph G1 instead of a single (simple) arithmetic node. End-point1 and End-point2 (where G1 is stitched into G) are of special interests. Boundary conditions [Chen91] are spelled out by the values of the pseudo-time labels T_{bound1} and T_{bound2}. The fundamental rationale is that no matter what transformations are carried out or how the pseudo-time labels are changed, the boundary conditions should be preserved. Mathematically, we have

$$T_{bound1} < T1$$
$$\max\{T1,T2\} < T3 < T_{bound2}$$

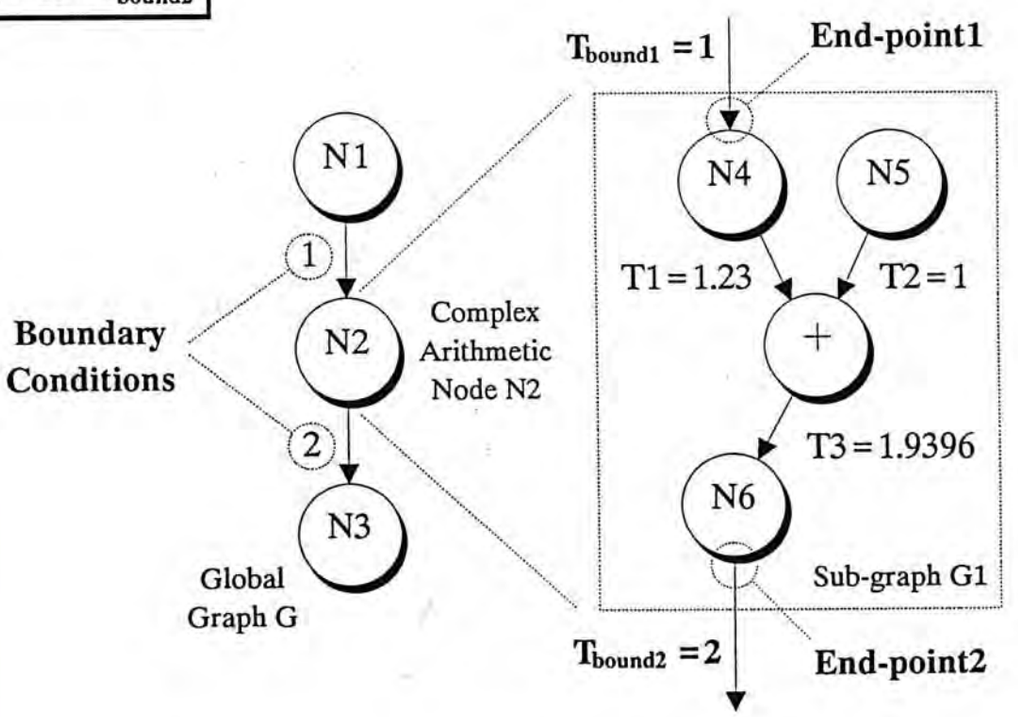


Figure 2.17. Non-integer pseudo-time labels

Suppose the data from N5 is immediately available/valid, therefore we can make T2=1, implying $T_{bound1} < T1 < T3 < T_{bound2}$. That is just what we have expected. In fact, when hardware implementation is concerned, a more realistic interpretation should be :

$T1 = T_{bound1} + \varepsilon_1$
$T3 = T_{bound1} + \varepsilon_1 + \varepsilon_2$

where $\varepsilon_1, \varepsilon_2 > 0$ and $\varepsilon_1 + \varepsilon_2 < 1$. By substituting the values of T_{bound1} and T_{bound2}, we have $1 < T1 < T3 < 2$. As a result, both T1 and T3 can assume real numbers only, as exemplified by figure 2.17.

In this sense, the use of real numbers for pseudo-time labels seems unavoidable. Yet, their (hardware) representation will be inexact. It becomes intuitively difficult to manipulate, compare and normalize time labels. For example, given any arbitrarily small $T > 1$, there exists $T' \in \mathbf{R}$ (the set of real numbers) satisfying $1 < T' < T$. Here we propose a more elegant scheme which involves the use of multi-level pseudo-time labels, each assuming the following format :

$p_1 \cdot p_2 \cdot p_3 \cdots p_{n-1} \cdot p_n$

where $p_i \in \mathbf{N} \cup \{0\}$ ($\mathbf{N}$ being the set of natural numbers and thus each p_i is a non-negative integer) for $i = 1,2,3,...,n$. The rule of comparison is modified accordingly. Given two multi-level pseudo-time labels $T = p_1 \cdot p_2 \cdot p_3 \cdots p_n$ and $T' = q_1 \cdot q_2 \cdot q_3 \cdots q_m$, with $q_i \in \mathbf{N} \cup \{0\}$ also for $i = 1,2,3,...,m$. Suppose $p_i = q_i$ for $i = 1,2,3,...,s$ where $0 \le s \le n$ and $0 \le s \le m$, we say that $T < T'$ if

$s = n$ and $s < m$

or

$p_{s+1} < q_{s+1}$

Thus we have $1.2 < 1.2.3$ and $1.2.3 < 1.2.4$. With multi-level pseudo-time labels, the concept of abstraction is implemented. A time label $T = p_1 \cdot p_2 \cdot p_3 \cdots p_{n-1} \cdot p_n$ is said to

be at the n-th level with its boundary condition spelled out by $p_1 \cdot p_2 \cdot p_3 \cdots p_{n-1}$ and its local precedence constraint encoded by p_n. As an illustration, we refer to the earlier example in figure 2.17 again and modify T1, T2 and T3 as follows :

T1=1.1
T2=1.1
T3=1.2

Figure 2.18. An example illustrating the use of multi-level pseudo-time labels

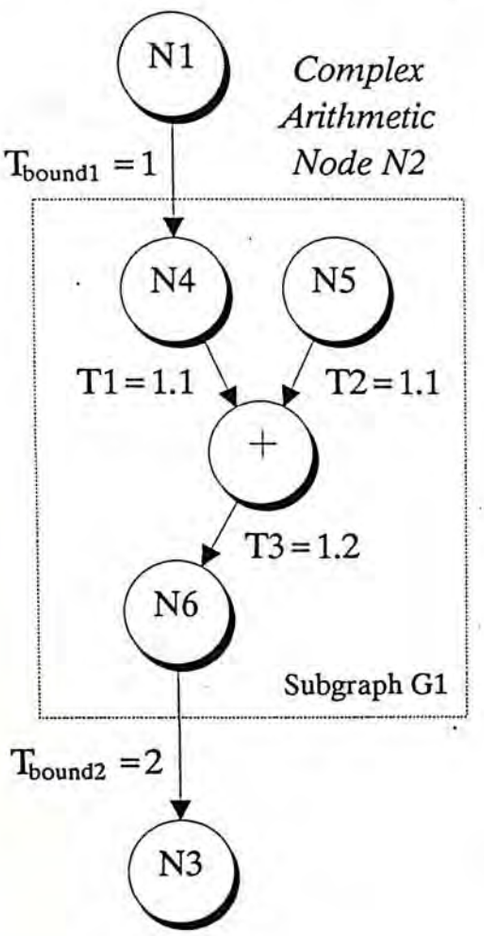
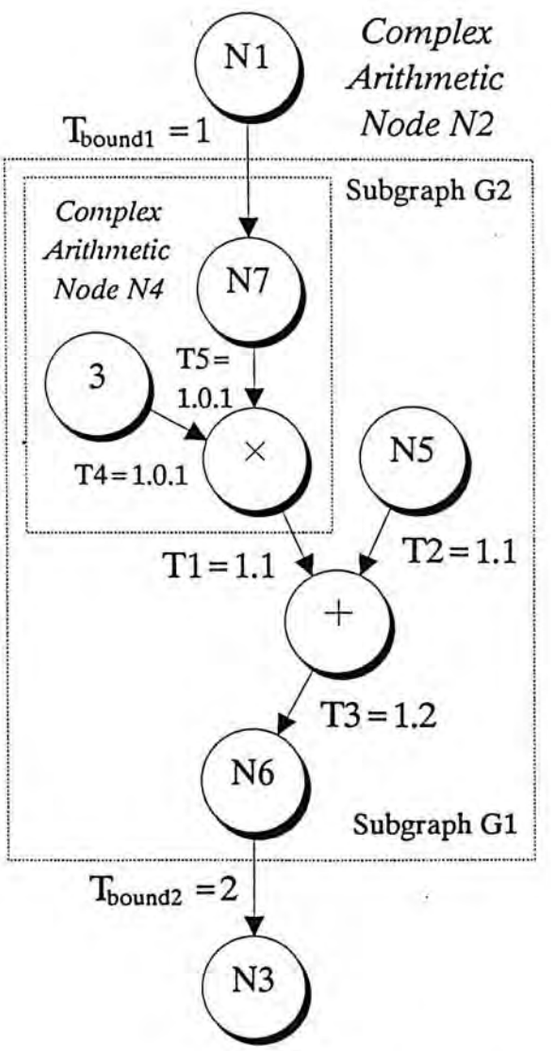


Figure 2.19. Multiple levels of subprogram abstraction



The resulting situation is depicted in figure 2.18. The leftmost "1" in both "1.1" and "1.2" records their respective lower boundary condition T_{bound1}. At the same time, the upper boundary condition $T1,T3 < T_{bound2}$ is also implied naturally. By having p_n of

T1 smaller than p_n of T3, the local precedence constraint T3 > T1 (within the sub-graph G1) is satisfied.

The use of multi-level pseudo-time labels facilitates the discussions (and descriptions) of subprograms. Further, by substituting "i" for the boundary condition $p_1 \cdot p_2 \cdot p_3 \cdots p_{n-1}$, different values of "i" would correspond to different calls to the subprogram. Events and operations within different calls are (globally) ordered with respect to their respective order of subprogram invocations. The idea of abstraction can be generalized to arbitrary number of levels. As illustrated by figure 2.19, N4, being a complex arithmetic node also, manifests another subgraph (sub-algorithm) G2 which is nested inside G1.

The implication of having each pseudo-time label made up of two components - the boundary condition and the local precedence constraint, is non-trivial. We can now transform a subgraph or manipulate its time labels independently without worrying for the boundary conditions. At the same time, changing the boundary conditions would not interfere with the local precedence relationship also. Take for instance, suppose the value of T_{bound1} in figure 2.18 is now equal to 2. In response, we only have to modify T1 to 2.1, T2 to 2.1 and T3 to 2.2, leaving their respective p_n unchanged.

2.5 Procedure Graph Optimizations

The identification of parallelizable operations is a dual problem. First, we need a scheme to encode the precedence constraints among the candidate instructions. Then optimization rules are applied to move the operations back and forth to arrive at an efficient schedule.

2.5.1 Representing Dependencies

We believe that causality preservation is one of the major burdens of performance improvement. The concept of different dependency relationships have been widely discussed in the literature (see [Hennessy&Patterson90] and [Padua&Wolfe86]). Casually speaking, they can be classified into four categories. In succeeding discussions, we will see how they are represented using procedure graphs. Conventionally, the precedence relationship involved is transparent, though affecting the sequencing of operations. But to facilitate discussion and devising ways to override this precedence relationship, the procedure graph theory chooses to make it explicit with its time labels.

First, Artificial Dependency, as its name suggests, refers to unnecessary ordering in the executions of instructions as implied by a sequential programming language. As a consequence of the "von Neumann bottleneck" of having only one single program counter, traditional programmers are dictated (or used to) think sequentially, and then code sequentially. Often, chances of parallel or overlapped processing are overlooked and the maximum performance is sustained. The example of Gaussian elimination inner loop in figure 2.7(b) serves as a good illustration.

On the other hand, when limited resources are shared, conflicts can occur. Let's examine the case in figure 2.20 where we have three multiplications each ready for execution immediately. Suppose there is only one multiplier available. The natural consequence is that these three multiplications will be forced to queue for the single multiplier in a strict sequential manner. Even worse, the addition F1←F1+1, though is ready for execution immediately, turns out to be unnecessarily held as the instruction fetch unit and the decoder would have been stalled by the structural dependency involving the multiplier upon handling the second multiply instruction. In this sense, resource conflicts or structural dependencies seem to be unavoidable at first glance. Degree of overlapped processing of instructions becomes dictated by resource availability as a result.

Procedural or Control Dependency deals with the dynamic overriding of the sequential top-down execution sequence of consecutive instructions because of conditional or unconditional branches. In the worst case, "bubbles" are injected into the instruction pipeline during the branch delay period (the time from the branch is decoded to its evaluation completes) until the correct target is known[4].

(a) Code Sequence

```
LD      F0,A[1]
MUL     F0,@3
ST      A[1],F0
LD      F0,A[2]
MUL     F0,@3
ST      A[2],F0
LD      F0,A[3]
MUL     F0,@3
ST      A[3],F0
ADD     F1,@1
```

(b) Procedure Graph Representation

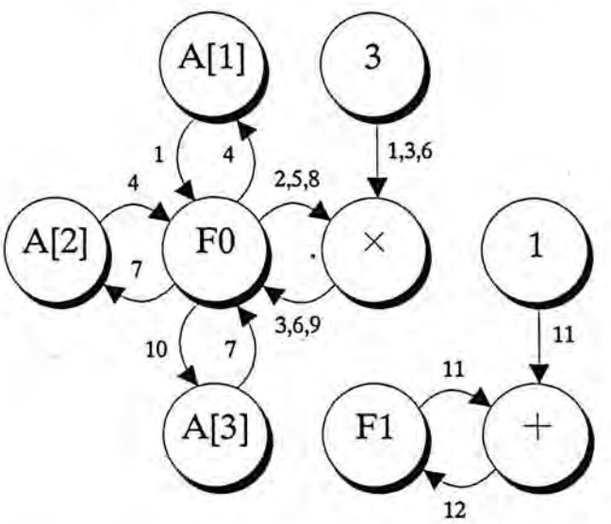


Figure 2.20. An example showing the effects of structural dependency (resource conflict)

Finally, Data Dependency occurs when an access to a storage location is being locked by another access involving the same storage location. Following the classification and notation of H. Stone [Stone87], typical data dependency phenomena include Read-After-Write dependency (RAW), Write-After-Read dependency (WAR) and Write-After-Write dependency (WAW). Figure 2.21 shows their corresponding procedure graph representations. Respective examples are also given for illustration.

With reference to figure 2.21(a), the right operand of instruction S2, that is, the content of F0, will not be ready until S1 completes. As a result, the execution of S2 must follow S1, We say that there is a RAW dependency from S1 to S2, written as S1 δ S2. On

[4] We choose to postpone discussing the procedure graph representation of procedural dependency after the introduction of certain new constructs in the next chapter.

the other hand, the situation is different in figure 2.21(b). Now, the original content of F0 (delivered from A[1]) should be preserved until S2 has completed its read-access to it. Therefore, S2 should precede S3 in execution, denoted by S2 $\bar{\delta}$ S3. Finally in figure 2.20(c), S3 must follow S2 lest F0 and F2 will contain the wrong values. Note that the constraint will still hold even if S2 is absent, but then S1 can be dropped altogether (by applying a Store-Store Cancellation). The three types of data dependencies are also commonly referred as True data dependency (or Flow dependency), Antidependency and Output dependency respectively (see [Padua&Wolfe86]). Although the original discussions of these dependency cases treat an instruction as indivisible (atomic), one can have RAW, WAR and WAW dependencies in instruction fragments also.

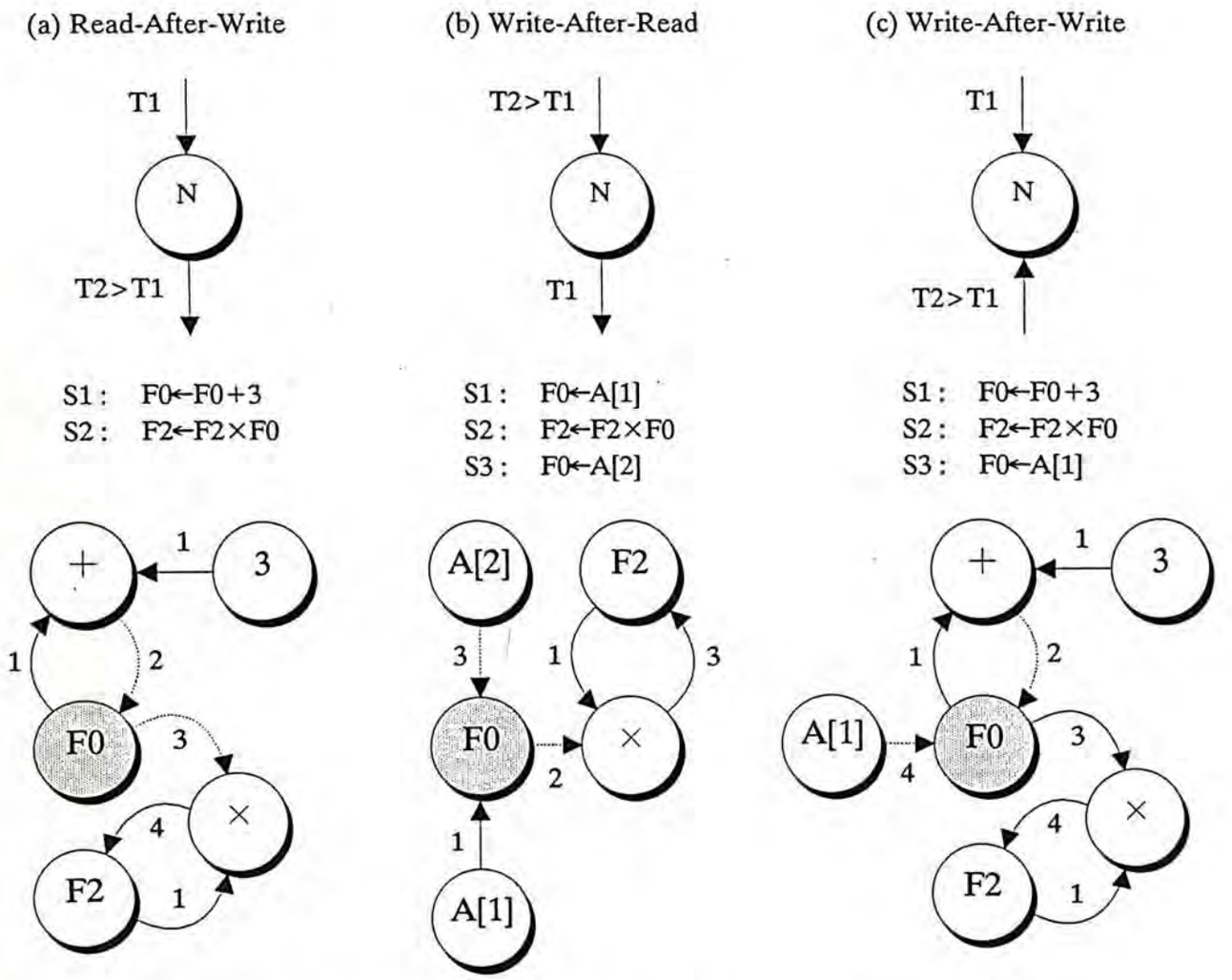


Figure 2.21. Procedure graph representations of different types of data dependencies

2.5.2 Eliminating Unnecessary Dependencies

The existence of any kind of dependencies all lead to a single consequence - a certain execution order is being dictated for the set of operations to perform, and a "loyal" CPU is constrained to serialize operations. But in some cases, some of the dependencies are in fact unnecessary and the order prescribed may not be what the particular computation demands. Whether this is the responsibility of the machine architecture, the compiler which generates the machine instructions or the programmer who codes the algorithm, such a strict order of execution is undesirable, since possible potential of overlapping and/or out-of-order executions would be overlooked (when the precedence or causality is in fact irrelevant).

Regardless of the cause, the causality required implies a global precedence relationship among each event in the computer - something cannot happen until another thing has finished.

Most of the dependencies mentioned in the preceding section are in fact unnecessary and honoring them would decrease the overall performance. Clever architecture designs should try to avoid them. Every precedence representation scheme should be accompanied by a mechanism for identifying parallelizable operations. This is easily observed in procedure graph theory and equivalent graph transformation is the most powerful tool.

First, structural dependency as dictated by the availability of resources can be resolved by appropriate "resource duplication". By adopting reservation stations (see [Johnson91]), operations queueing for certain shared resources (e.g. functional units) to become free can be buffered so that succeeding independent instructions in the sequential instruction stream can be examined. Hidden chances of parallelism becomes uncovered as a result.

As an illustration, let's return to our earlier example in figure 2.20. The adoption of reservation stations effectively redefines an arithmetic node as a complex node consisting of several temporary nodes, each being capable of buffering one operation. As exhibited in figure 2.22, by using three multiply reservation stations, the three multiplications are effectively buffered. Although they still have to queue for the single multiplier, the ADD instruction is no longer blocked from issue and can now proceed immediately.

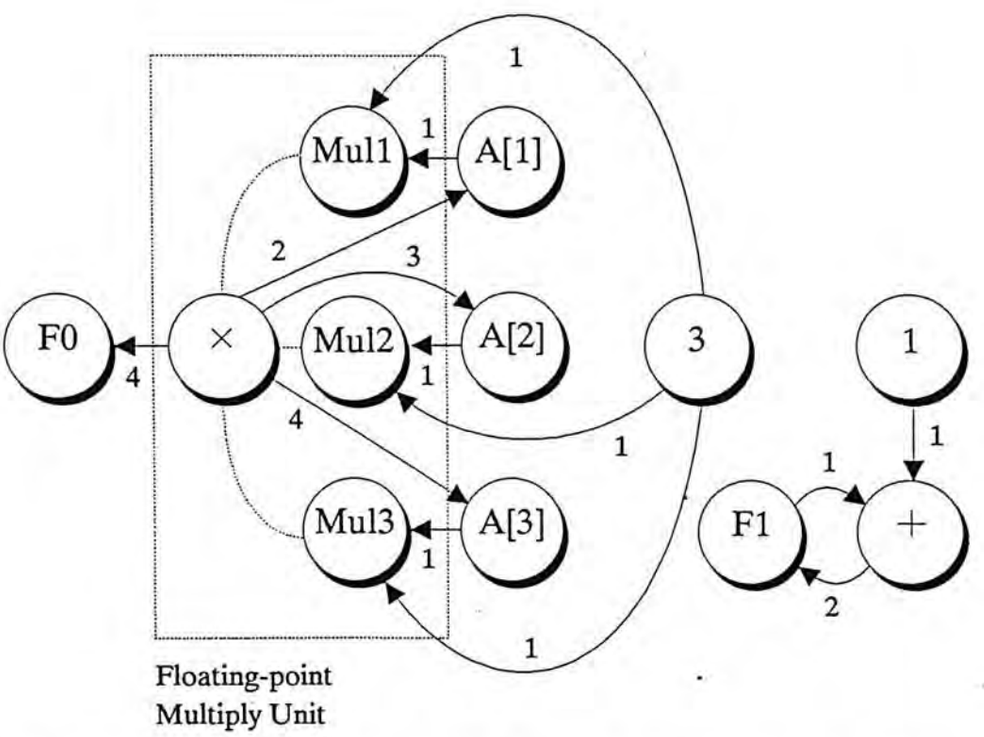


Figure 2.22. With reservation stations, the ADD is no longer blocked from issue

(× : Multiplier; Mul1, Mul2 and Mul3 : Multiply Reservation Stations)

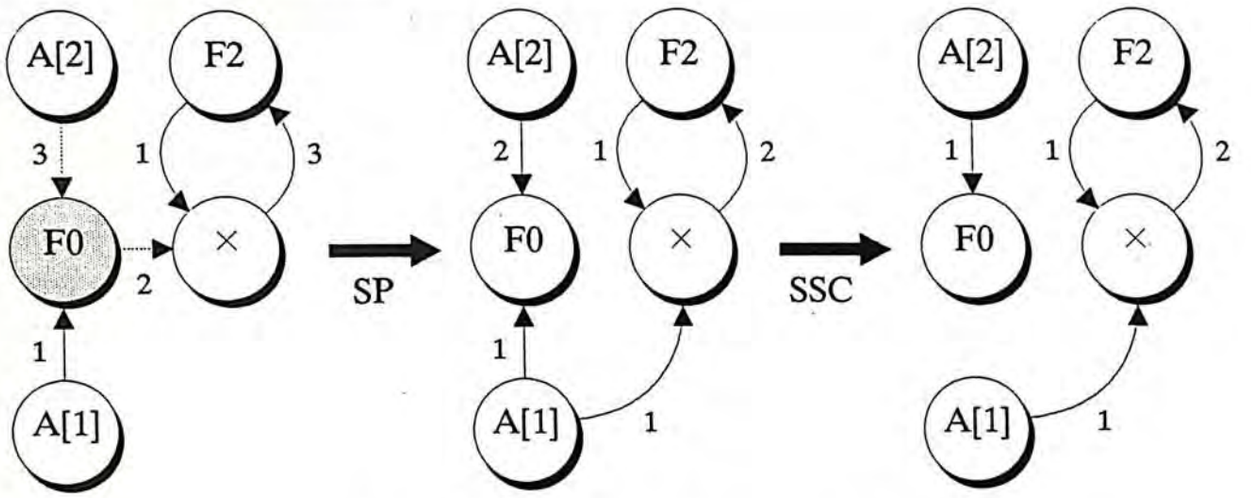


Figure 2.23. Resolving WAR dependency

(SP : A[1] F0 × ; SSC : A[1] F0, A[2] F0)

On the other hand, the WAR dependency depicted in figure 2.21(b) can be resolved by applying the transformation sequence SP(A[1] F0 ×) followed by SSC (A[1] F0, A[2] F0), as shown in figure 2.23.

In much the same way, the WAW dependency (involving the register F0) in figure 2.21(c) can also be overridden using equivalent graph transformations. As depicted in figure 2.24, the output of the add instruction S1:F0←F0+3 is forwarded to the multiplier directly and only the load instruction S3:F0←A[1] writes to F0. As a result, they can proceed in an overlapped manner.

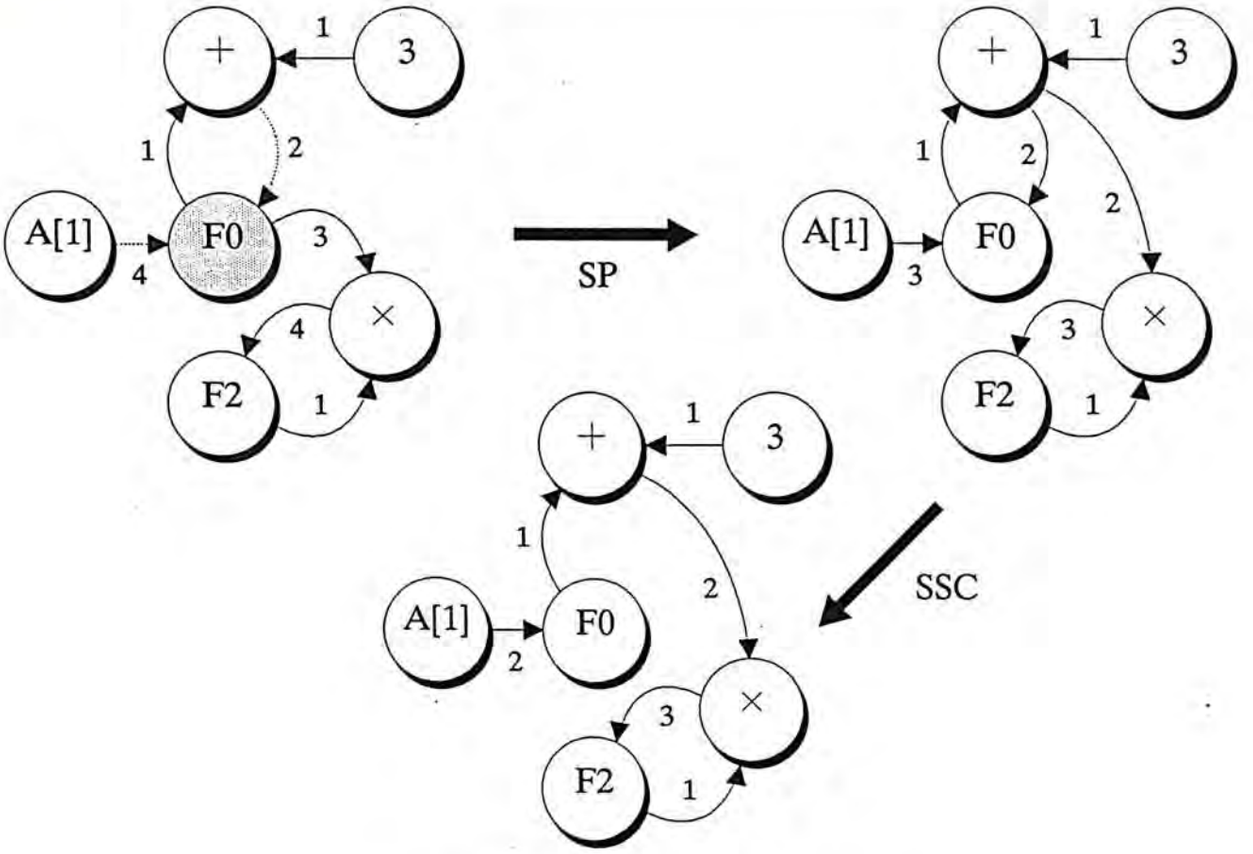


Figure 2.24. Overriding WAW dependency

(SP : + F0 × ; SSC : A[1] F0, + F0)

However, RAW dependency as related to the availability of data is unavoidable as dictated by the way the solution algorithm is formulated (not necessarily how it is coded). In particular, a piece of data cannot be consumed (say, being read as an operand) until it is produced (e.g. as the output of an arithmetic instruction). Significant speedup of a "sequential" program cannot be realized without a re-structuring of the solution/algorithm itself (see [Hillis&Steele86]).

The existence of RAW dependency also borders the handling of procedural dependency as the outcome of a branch instruction often depends on the computation results of some preceding instructions (which still haven't finished their executions at the time the branch instruction is considered). Several techniques have been successful in tackling the problem. Some of them are applied at the software level as a part of the compiler optimizations (e.g. delayed branching [Hennessy&Patterson90]). Others are

incorporated in the machine architecture (e.g. speculative execution [Smith et al.90]). In chapter 3 and chapter 5, we will discuss them in greater detail.

2.6 Simulation Program

A simulation program on procedure graphs and transformations has been implemented in the Sun Sparc Workstations. Actually, the system is composed of two parts (or two independent processes communicating using "sockets"). With its graphical interface (built in SunView), the user can now specify a procedure graph conveniently using the "interface" part (which is written in C) of the system, and then the "processing" part (which is written in Prolog) will derive its equivalences one-by-one interactively. Graph transformations are achieved via the three classical internal forwarding rules : Serial-to-Parallel Transformations (SP), Parallel-to-Serial Transformations (PS) and Store-Store Cancellations (SSC). We believe that this simulation program is useful for testing new equivalences identified as well as evaluating different execution strategies.

2.6.1 Preliminary Study Using the Simulation Program

Some experiments have been carried out using the simulation program. A particular example involving the Gaussian elimination inner loop is studied (using one element only). A total of 26 equivalent graphs are identified which have been drawn in figure 2.25. The one numbered 0 is the original graph as suggested by the machine code generated by a FORTRAN compiler (see [Chen91]).

Using these 26 equivalences as nodes, another graph has been drawn in figure 2.26, with directed edges denoting the transformation between graphs, and labels on them denoting the transformation rules applied. The results do reveal some of the interesting characteristics of procedure graphs such as parallelism and concurrency in derivations/transformations as well as the combinatorial explosion of procedure graphs. In succeeding discussions, we will discuss them in detail.

2.6.2 Economic Factors

Graphs obtained via transformations are mutually equivalent, in the sense that they give rise to the same set of results upon identical inputs, though implying different performance, cost, realizability, etc. From a practical point of view, some kind of evaluation criteria should be formulated to guide a wise choice. These criteria, referred as economic factors, may include the followings [Chen91] :

- number of memory accesses required
- number of arcs involved
- total time duration
- number of nodes involved

As an illustration, information concerning the first three factors have been incorporated in figure 2.25 for reference. The number of nodes remains as 5 in all graphs, and thus is omitted.

Attention should be drawn to a couple of things. First, the determination of the relative weights of various economic factors may simply turn out to be a matter of personal perception only. Subjective opinion is thus unavoidable.

In addition, the effects of these factors very often oppose each other. A transformation favouring one factor may sacrifice another at the same time.

More importantly, while the notion of equivalence is permanent, the value of a so called economic factor can change over time, say, because of technological advances.

Finally, easy realizability may turn out to be the major consideration when implementation is concerned (commonly applied decision criteria may include fanout

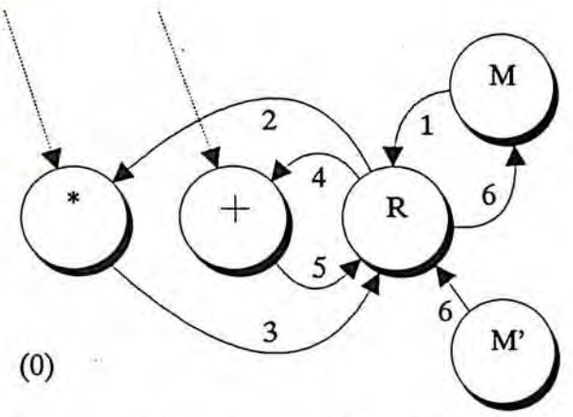
limitation, hardware complexity, machine instruction format, etc). All these make a quantified evaluation difficult to apply.

2.6.3 Combinatorial Explosion of Procedure Graphs

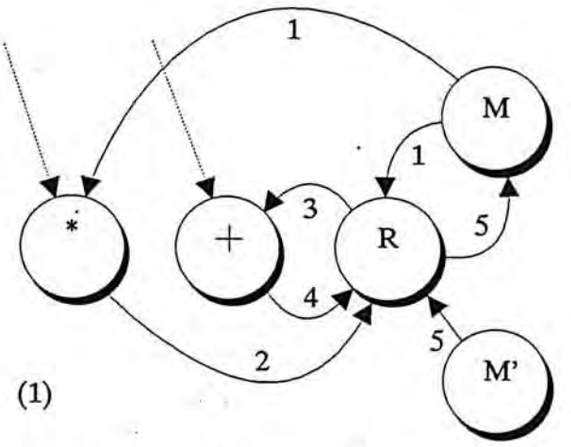
A simple procedure graph can give rise to a large number of equivalent graphs. As an example, consider again the procedure graph of the Gaussian elimination inner loop involving two elements. Surprisingly enough, more than 800 equivalent graphs are derived by our simulation program. Moreover, the result is obtained using the three classical internal forwarding techniques only. Thus it can be expected that the actual total number of equivalences may be even larger. This demonstrates the great generative power of procedure graphs by adopting pseudo time labels. We believe that this is only one of the many interesting mathematical properties of procedure graphs, which should merit more study effort.

Figure 2.25. Equivalent graphs of the Gaussian elimination inner loop of 1 element only

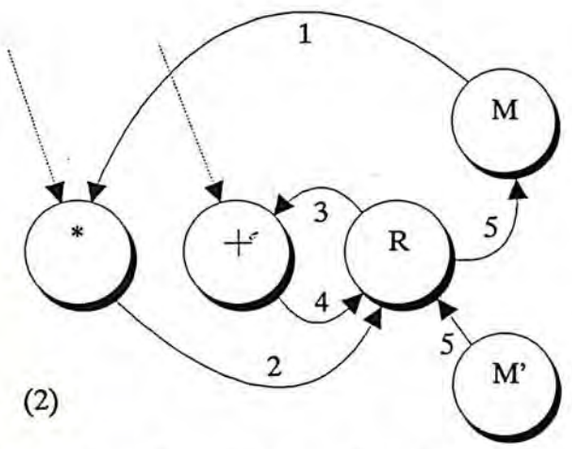
(to be continued)



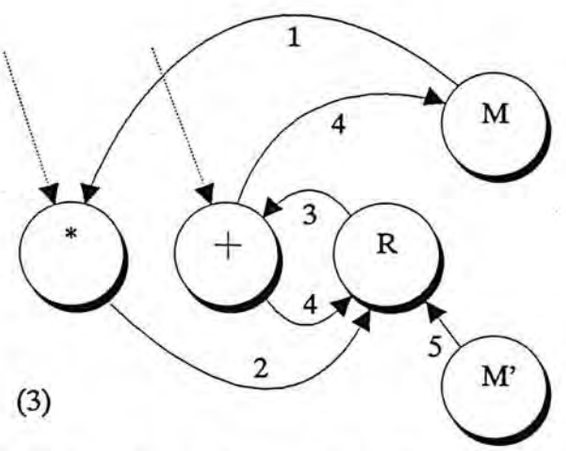
Duration=6 Memory=3 Transfers=7



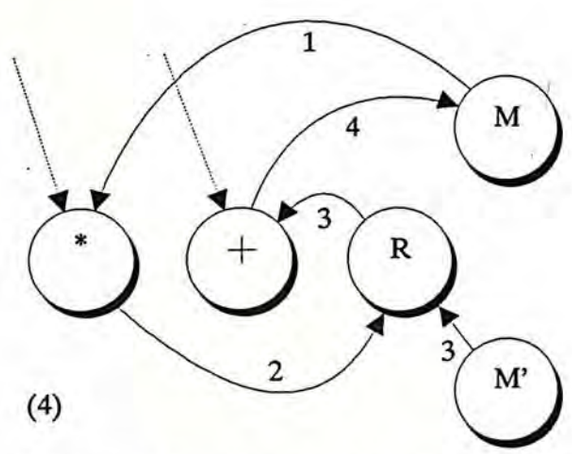
Duration=5 Memory=4 Transfers=7



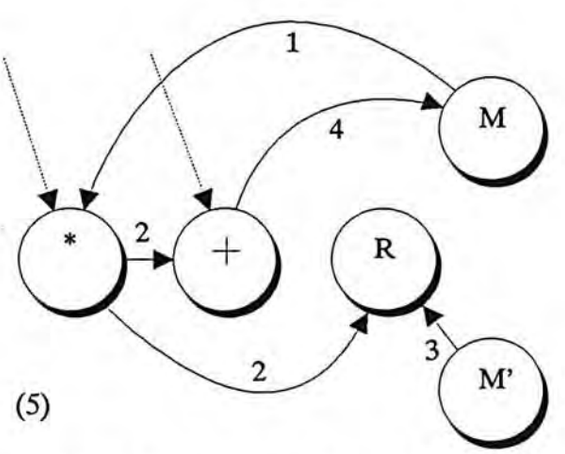
Duration=5 Memory=3 Transfers=6



Duration=5 Memory=3 Transfers=6



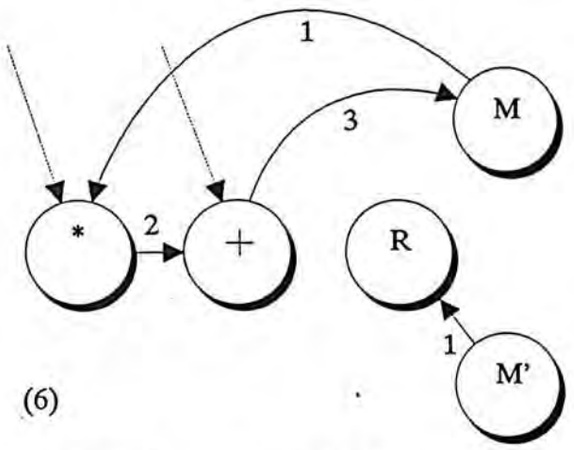
Duration=4 Memory=3 Transfers=5



Duration=4 Memory=3 Transfers=5

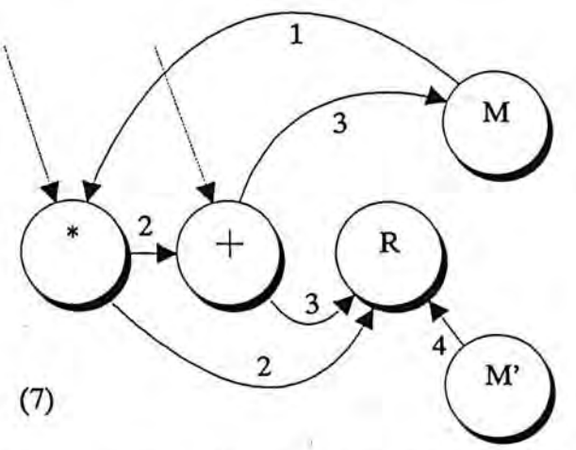
Figure 2.25. Equivalent graphs of the Gaussian elimination inner loop of 1 element only

(continued)



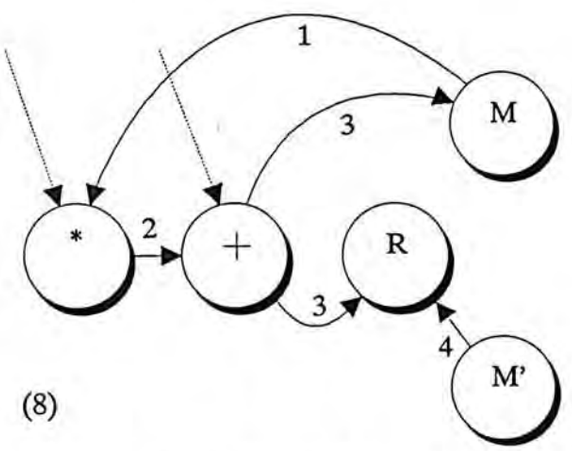
(6)

Duration=3 Memory=3 Transfers=4



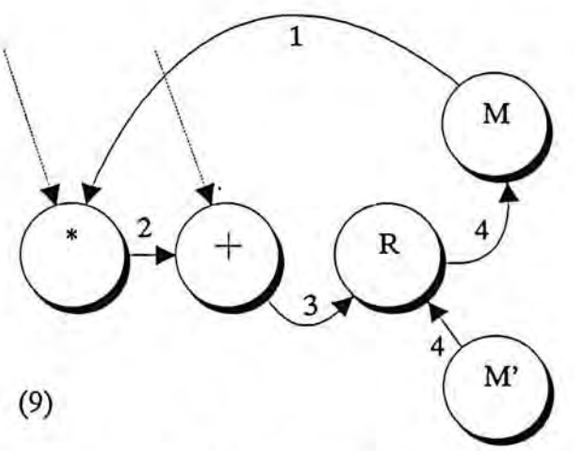
(7)

Duration=4 Memory=3 Transfers=6



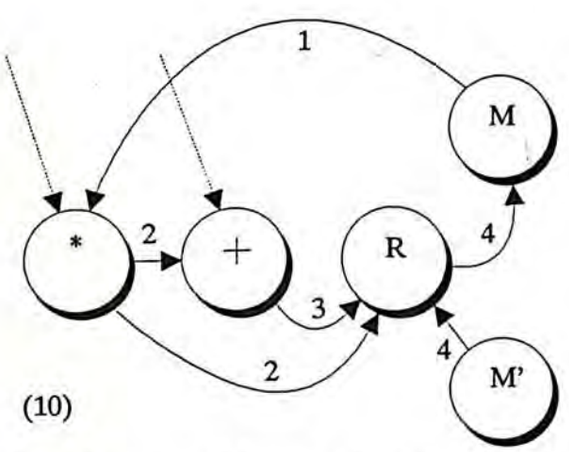
(8)

Duration=4 Memory=3 Transfers=5



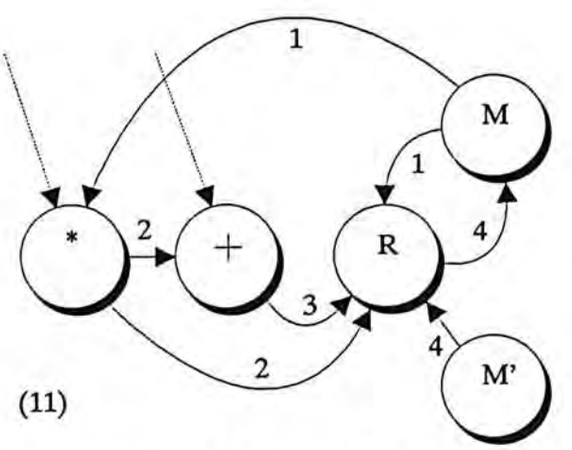
(9)

Duration=4 Memory=3 Transfers=5



(10)

Duration=4 Memory=3 Transfers=6

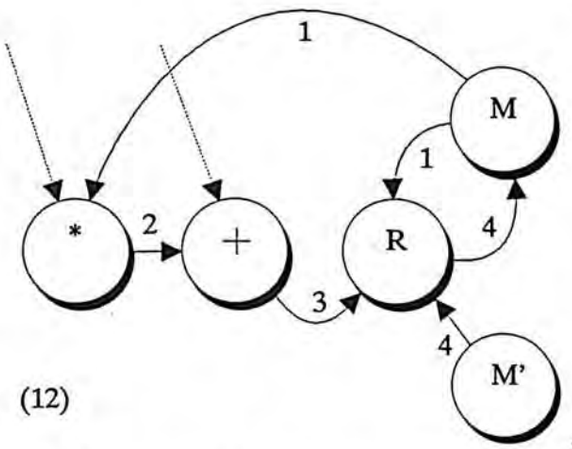


(11)

Duration=4 Memory=4 Transfers=7

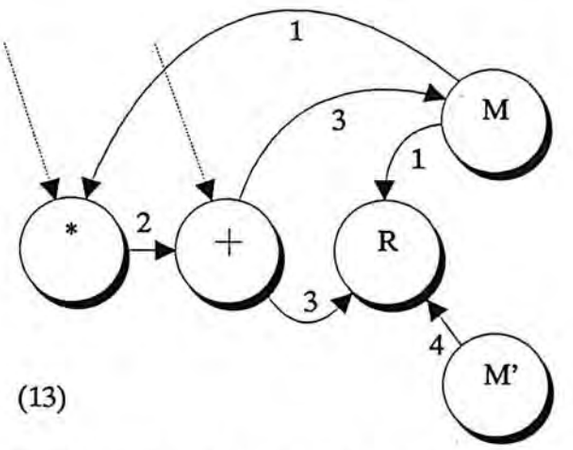
Figure 2.25. Equivalent graphs of the Gaussian elimination inner loop of 1 element only

(continued)



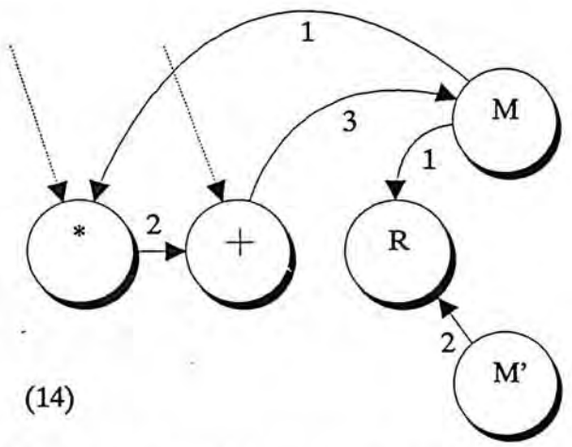
(12)

Duration=4 Memory=4 Transfers=6



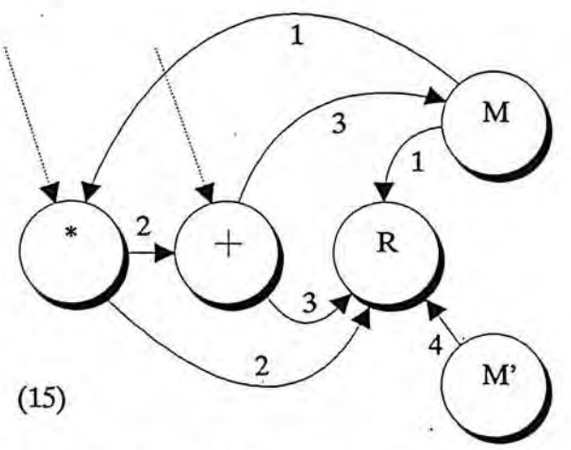
(13)

Duration=4 Memory=4 Transfers=6



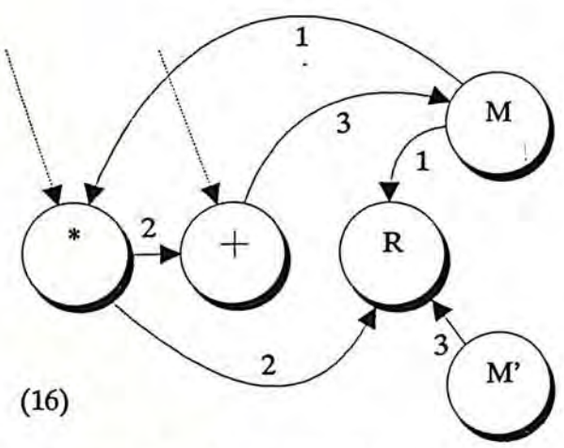
(14)

Duration=3 Memory=4 Transfers=5



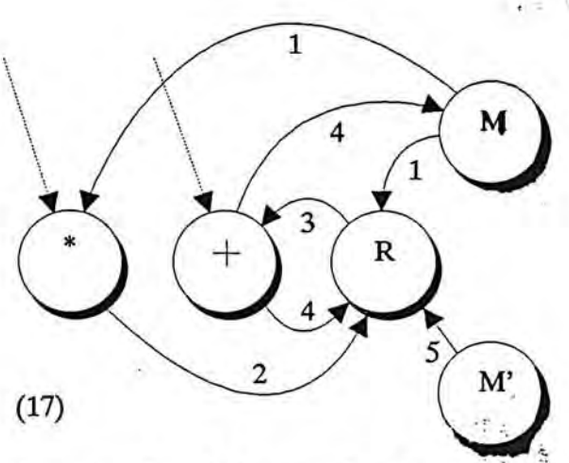
(15)

Duration=4 Memory=4 Transfers=7



(16)

Duration=3 Memory=4 Transfers=6

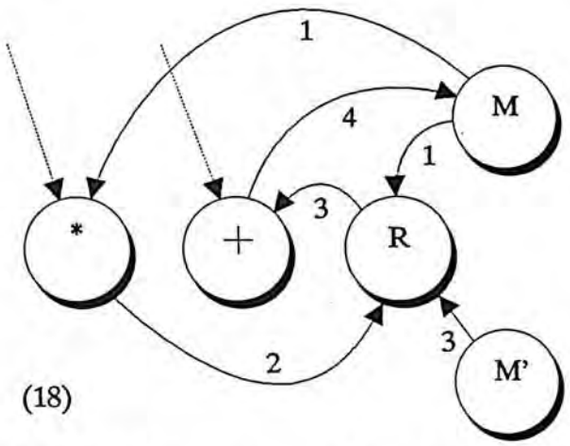


(17)

Duration=5 Memory=4 Transfers=7

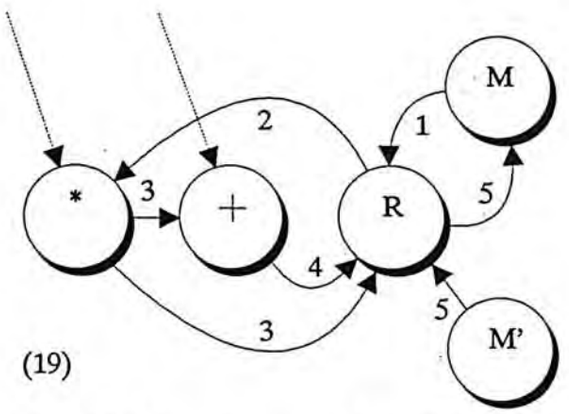
Figure 2.25. Equivalent graphs of the Gaussian elimination inner loop of 1 element only

(continued)



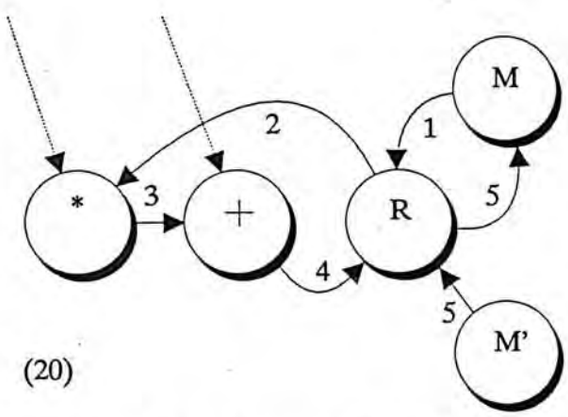
(18)

Duration=4 Memory=4 Transfers=6



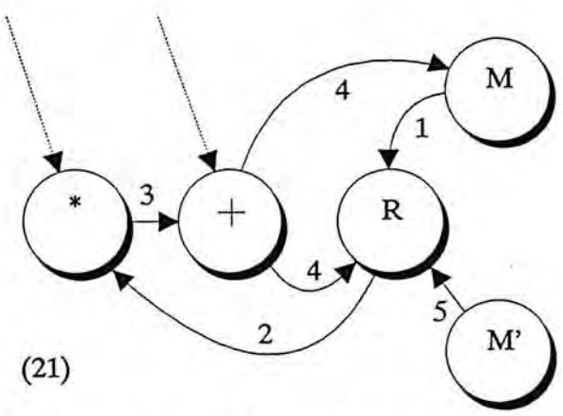
(19)

Duration=5 Memory=3 Transfers=7



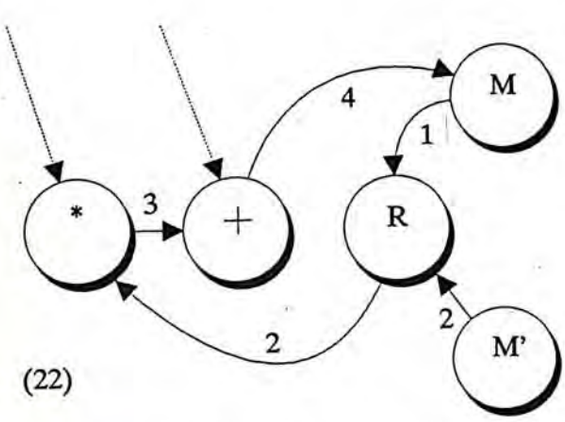
(20)

Duration=5 Memory=3 Transfers=6



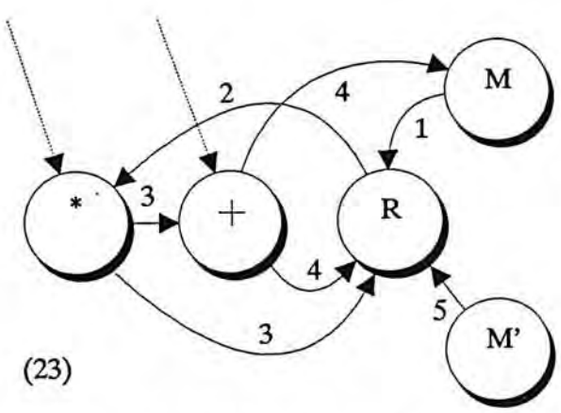
(21)

Duration=5 Memory=3 Transfers=6



(22)

Duration=4 Memory=3 Transfers=5

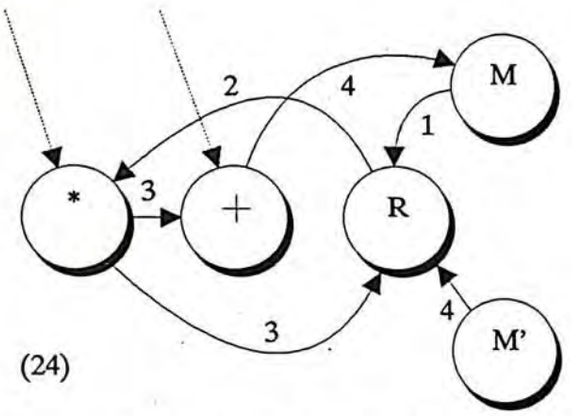


(23)

Duration=5 Memory=3 Transfers=7

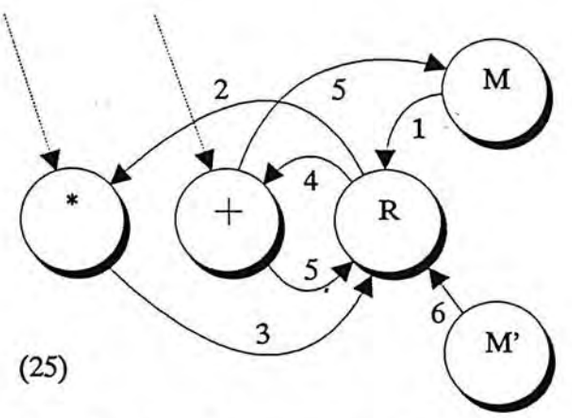
Figure 2.25. Equivalent graphs of the Gaussian elimination inner loop of 1 element only

(continued)



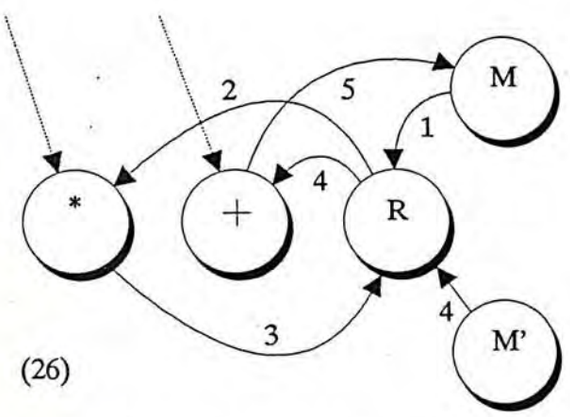
(24)

Duration=4 Memory=3 Transfers=6



(25)

Duration=6 Memory=3 Transfers=7



(26)

Duration=5 Memory=3 Transfers=6

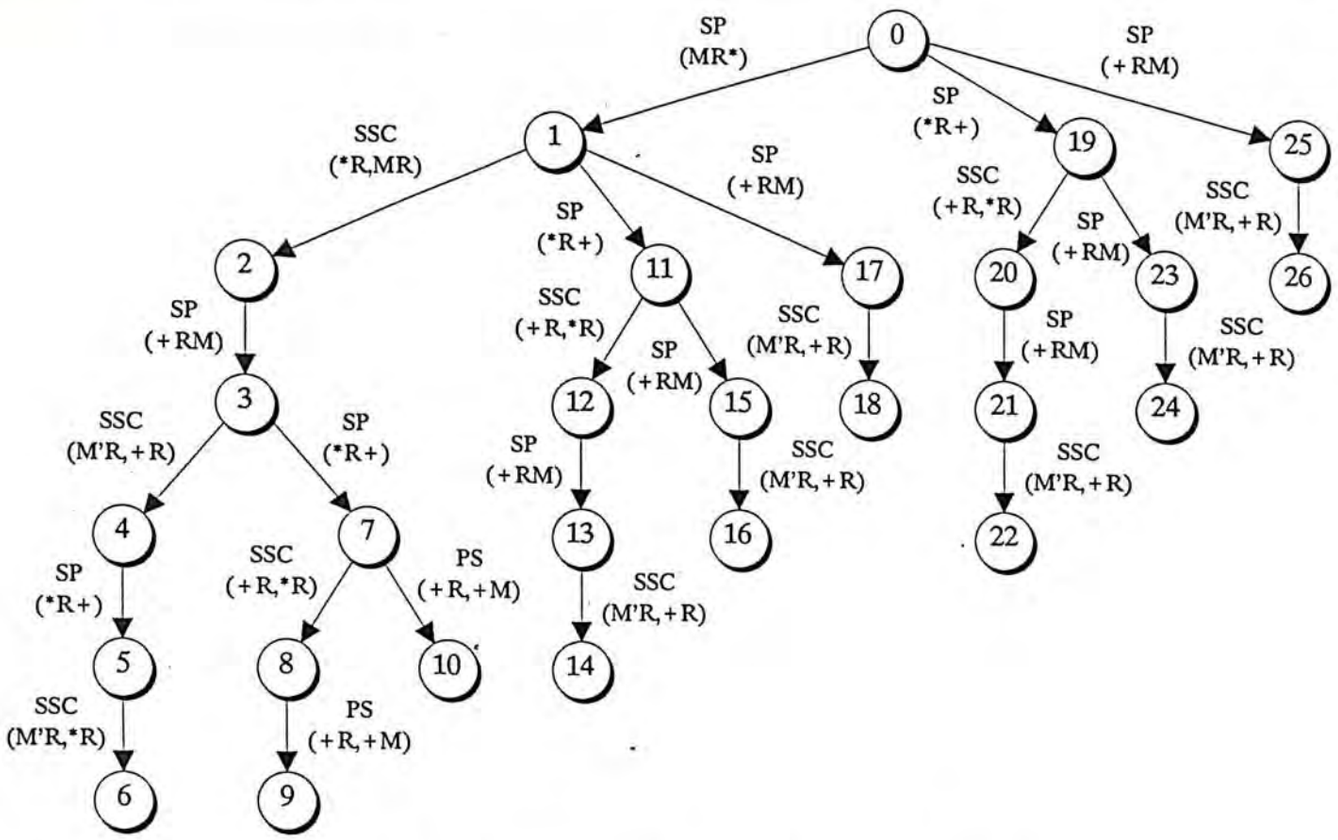


Figure 2.26. Transformations among equivalent graphs

Chapter 3 Extending the Procedure Graph Theory

3.1 The T-Operator and the F-Operator

So far in our discussion, data transfers manifested by labelled arcs are invoked sooner or later unconditionally. But occasionally, we may want some of these data transfers to be executed at the discretion of the presence or absence of certain conditions. More important, the choice should be made dynamically.

To incorporate such an element of uncertainty in our procedure graph model, two new constructs are introduced, namely the T-Operator and F-Operator, as depicted in figure 3.1. Basically, each produces a single numerical output with its two inputs : a numerical data (N) and a boolean data (B). Their semantics of operation should be obvious by examining the figure. Intuitively, the T-Operator behaves like a "permitter" while the F-Operator behaves like an "inhibitor". For clarity purposes, from now on, dashed arcs will be used to distinguish data transfers carrying boolean data.

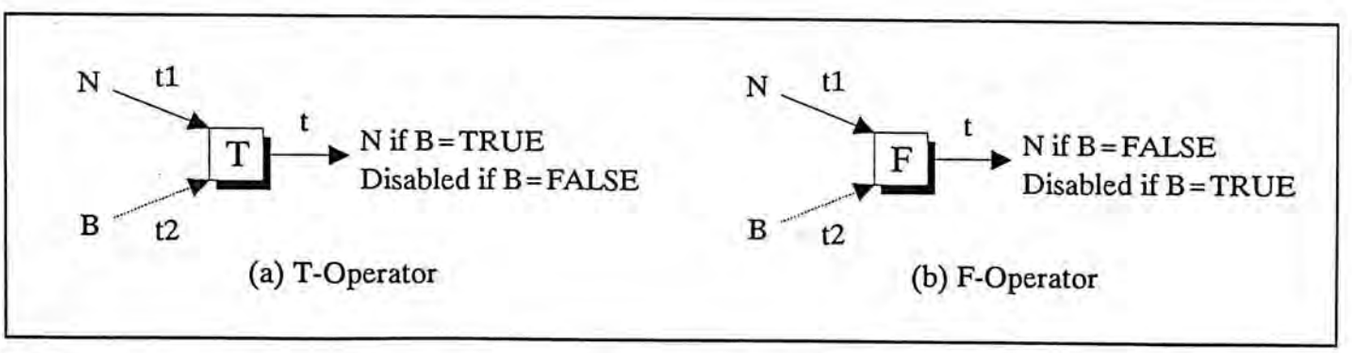


Figure 3.1. The T-Operator and the F-Operator (t > max{t1,t2})

An obvious benefit is that the two operators can facilitate the representation of programmer-specified conditional constructs similar to the following :

If α then β else δ

Sometimes, the accesses to certain T- and F-Operators may become privileged to system architects and designers only. In such circumstances, they do not constitute a part of the user's program under execution. Instead, their presence guarantees its correctness.

3.2 Modifying the Firing Rule

Still another modification to the Firing Rule of operator nodes is necessary. In the original procedure graph theory, operator nodes are given the greatest autonomy. Theoretically, an operator node is ready for firing as soon as all of its input operands are valid. The rationale behind is simple, namely, an instruction/operation with all of its true data dependencies resolved should not be unnecessarily blocked from execution.

However, this may pose serious problems in actual implementation. Imagine the case when we have 100 multiplications which are all ready for execution immediately, but only one multiplier available. Doubtless, some mechanism must be adopted to resolve the conflict.

With reference to figure 3.2, an extra input, the Fire Control, will be associated with each (user-referable) operator node as a dashed arc. As distinguished from ordinary data transfer arcs, this Fire Control carries a single bit of boolean data (or a pulse) instead of an operational data of full word-length. It won't take part in a computation as an operand. Rather, it initiates the firing of the operation and monitors its timing. Now, an operator with all of its input operands ready would still unable to proceed until triggered by a TRUE signal in the First Control (see [Chen91]).

Two uses of the Fire Control are illustrated in figure 3.2. In figure 3.2(a), the output of the multiplication is not ready until time=3. This ability to delay the firing of an operation is useful in resource-sharing and synchronization.

Please be noted that from this perspective, the new definition of operator nodes is in fact consistent with our choice to adopt reservation stations (see [Tomasulo67] and later discussions on the T-Architecture). Each pending operation is assigned to a free reservation station where they wait for its respective operands to become available. Every time when the operator or functional unit (say the adder) is free, operations with all operands valid will arbitrate for the shared resource (under a priority scheme or in a first-come-first-serve manner). A control signal will be sent to the winner to initiate its operation then. The reservation stations effectively serve as a buffer to uncover hidden parallelism.

In figure 3.2(b), we have simulated the following conditional construct :

```
        BZ    F1,@JUMP
        ADD   F1,F0
        ...   ...
@JUMP   ADD   F1,F2
```

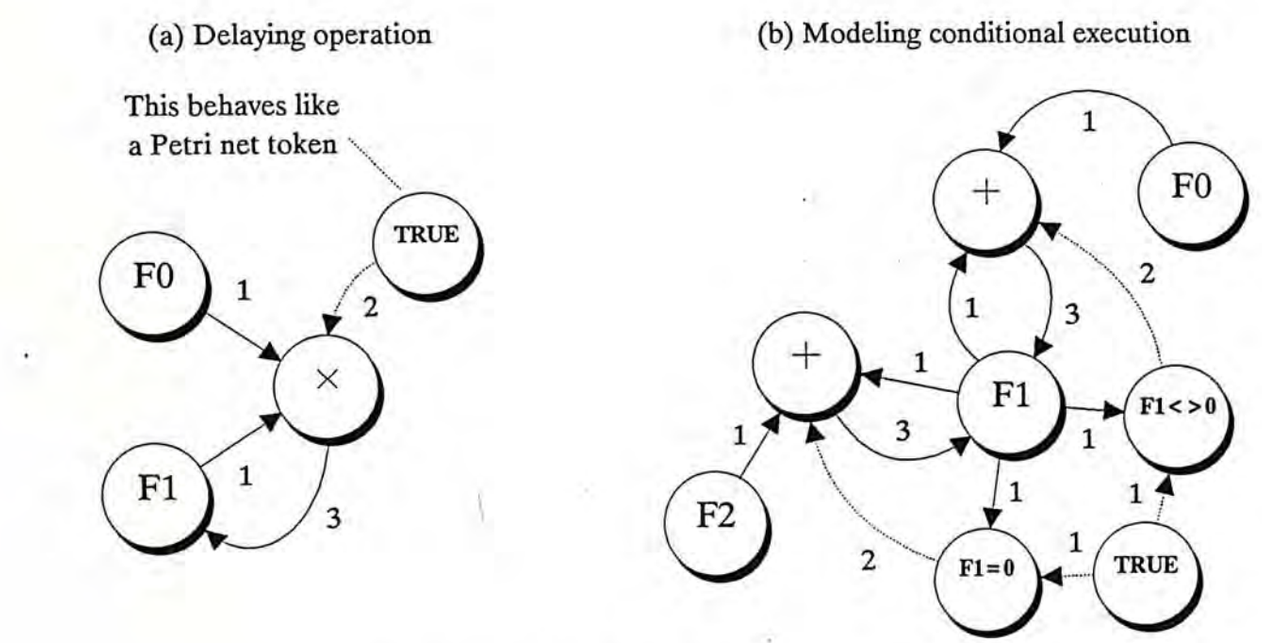


Figure 3.2. Revising the Firing Rule for operator nodes

As shown, both Add operators will be occupied by operands, yet, only one of them is fired. Thus, a mechanism should be adopted to release the other lest it will be left idle afterwards (a wastage of resources). This problem of "dangling input" makes the design unsatisfactory. Alternatively, we can also make use of the T-Operator and the F-Operator to obtain the "equivalent" procedure graph shown in figure 3.3. As shown, by making the Fire Control data-driven, the dynamic behavior of the system under investigation can be modeled more realistically.

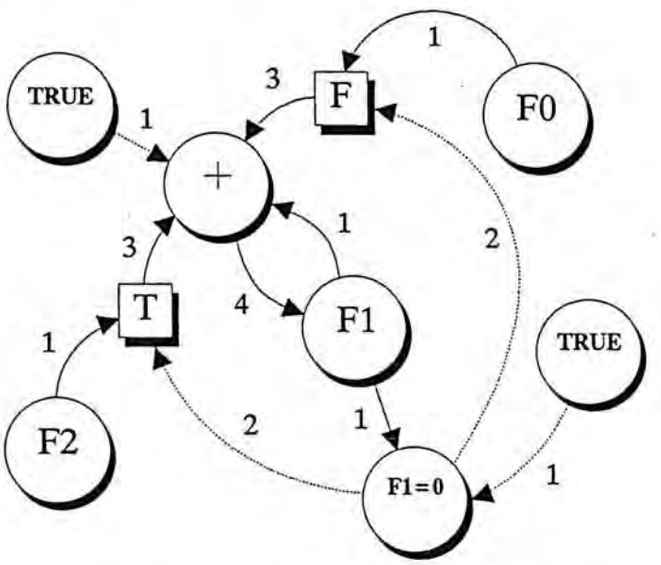


Figure 3.3. Simulating the conditional construct in figure 3.2(b) by T-Operator and F-Operator

To summarize, by adopting the T-Operator and F-Operator while modifying the Firing Rule, condition and negation can now be expressed neatly. The modeling power of the extended procedure graph theory will be increased as a result [Peterson81]. However, the transformation rules may have to be revised when the T-Operator and/or the F-Operator is/are involved.

Strictly speaking, the T- and F-Operators are not fresh new concepts as they can be simulated using existing constructs. Functionally, they resemble the decision node used in data flow computation. However, the fact that a decision node is a multiple-input, multiple-output entity/operator implies that its introduction in the procedure

graph theory may result in certain modifications to the fundamental concepts. Undesirable side-effects can be brought about. More discussions are thus necessary in order to evaluate the pros and cons involved.

As a final comment, we highlight here the possibility to generate the pseudo-time labels (that is, the precedence information) dynamically as outputs of certain "privileged" operations. The case should be interesting, we think.

3.3 Procedure Graph Representation For Different Branch Strategies

The presence of a branch instruction corrupts the sequential flow of control in a program. When the branch outcome depends on an uncompleted instruction under execution, the instruction fetch unit may have to be stalled and bubbles will be created in pipelines. Exactly, it is the approach adopted in figure 3.2(b) and figure 3.3. The net result is that the performance of a lookahead processor can be lost in the branch delay period (from the time the branch instruction is decoded to the time the address of the target instruction can be determined).

3.3.1 Multiple-Path Execution

Instead of waiting for the procedural dependency to be resolved, we can choose to continue execution along some predicted path(s) in a conditional mode. In a multiple-path execution environment, the two possible outcomes of every branch instruction (dependent on whether the branch condition is true or false) are considered separately. A "thread of control" is dedicated to "explore" each of them. In other words, conditional execution continues along multiple paths. Thus if N ($N>0$) branch conditional execution levels are allowed, the number of alternatives (or execution paths) will be 2^N. In the extreme case, all these possible outcomes are enumerated exhaustively. Sooner or

later the completion of a branch instruction will reveal that some of these trials are incorrect and/or unnecessary. In that case, those conditional executions will be terminated with all temporary computation results (if any) discarded.

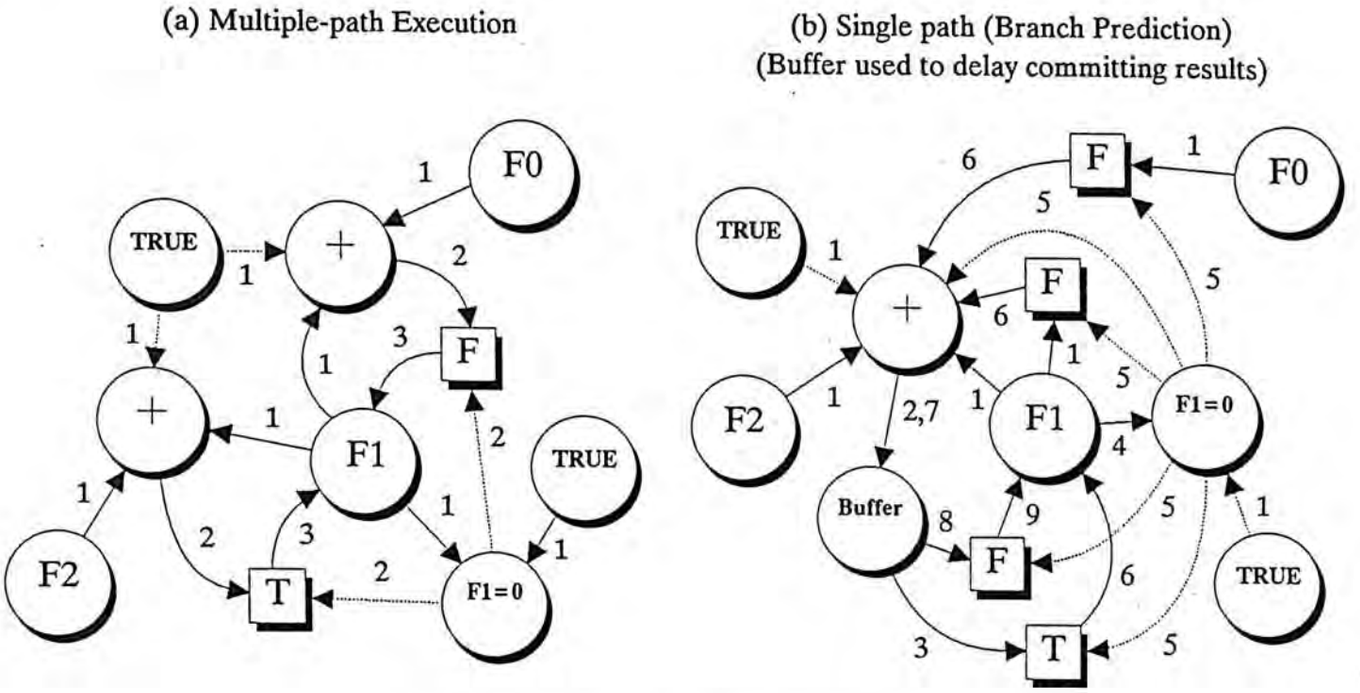


Figure 3.4. Conditional (or Speculative) Execution

Figure 3.4(a) serves as an illustration. Both additions are executed, and the evaluation of the BZ instruction "selects" the right one, effectively allowing only one addition to commit its result to the register F1 and discarding the computation result of the other.

Several drawbacks of this approach are obvious. First, multiple set(s) of computing resources (e.g. functional units, registers) should be provided to explore each possible execution path separately. On the other hand, control is also complicated when it becomes multi-threaded. And finally, combinatorial explosion will simply make multiple-path execution infeasible when the number of branch levels N is sufficiently large.

3.3.2 Conditional Execution with Delayed Commitment of Results

Thus to save hardware and simplify control, we choose one alternative only for each branch instruction encountered by guessing whether it will evaluate to true or false. Conditional execution then continues along the single predicted path. However, another problem is resulted. No matter how sophisticated the branch prediction mechanism is, there is always a chance that our prediction is wrong. When that's the case, we should be able to recreate the machine state just before the branch instruction and let the machine to restart from the correct target.

Still another choice should be made here. On the one hand, we can hold every attempt to modify the state of the machine, such as writing to a memory location, in a buffer until the corresponding branch dependencies have been resolved. Figure 3.4(b) illustrates the underlying mechanism. The conditional construct in figure 3.2 (and 3.3) is reproduced here, and branch prediction results in conditional execution being continued from the instruction "ADD F1,F2". Yet, no commitment of computation result will be allowed until the outstanding procedural dependency is resolved. Later if the evaluation of BZ reveals that our prediction for it was correct, the enabling of the T-Operator will update F1 using the computation result stored in the buffer. Otherwise, the T-Operator will be disabled, effectively discarding the wrong computation result. Execution then restarts from the correct target, that is, the instruction "ADD F1,F0".

However, several disadvantages make this approach unfavorable. Apart from the cost incurred in the provision of the buffer, additional overhead impedes every access. For example, we should be able to tell which one of the architectural register file or the reorder buffer stores the most updated value of a certain register. The situation is further complicated if multiple entries corresponding to the same storage location are allowed to coexist in the reorder buffer.

3.3.3 Speculative Execution with Register Backup and Branch Repair

On the other hand, we can take a snapshot of the machine's state just before the branch instruction and thereafter, allow modifications to be committed directly and immediately. Later, if the branch prediction turned out to be wrong in fact, the backup would be used to restore the correct machine state before the branch, effectively undoing the effects of all instructions executed along the predicted path. The scheme is primarily suitable for register data only because of the enormous size of the memory involved.

There exists a procedure graph representation for this register backup and branch repair mechanism. Upon a faulty branch, the ability to roll back to the correct machine state before the faulty branch represents, in some sense, a certain kind of external precedence constraint. Its application helps to restore the correct causality relationship, should it be found violated due to a mistakenly predicted (and executed) branch. The architecture provides the necessary backup and a mechanism to repair.

Let's consider the example in figure 3.5. Suppose that the floating-point ADD instruction takes three cycles to complete and that only one instruction can be fetched and decoded in each cycle. A natural consequence is that upon decoding the branch instruction (BZ), the content of the register F0 (being the destination of the ADD instruction) is not valid/ready yet. In fact, there is a true data dependency[1] from ADD to BZ. Therefore, the next instruction after the BZ instruction cannot be known.

As a result, the BZ instruction on the one hand calls for the branch prediction mechanism. By adopting the convention that each branch, when first encountered, is

[1] Denote, for an instruction i, the set of storage locations read and written by R(i) and W(i) respectively. We say that there is a *True Data Dependence* from an instruction i to another instruction j, written as $i\delta j$, if $W(i) \cap R(j) \neq \varnothing$ (see [Pauda&Wolfe86]).

assumed to be taken, conditional execution continues from the instruction labelled by '@10'. On the other hand, we need to make a backup copy of the value of each register just before the branch instruction. This constitutes a part of the correct state for the machine is to be rolled back to, should a branch misprediction occur.

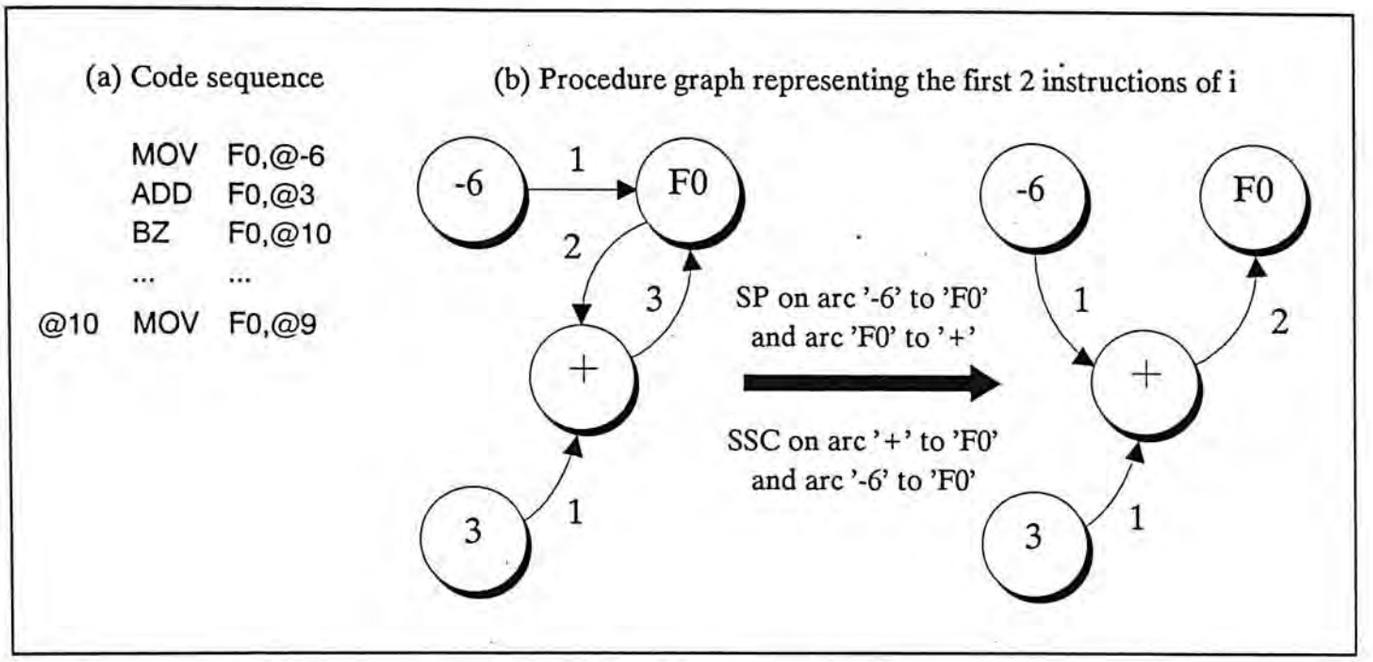
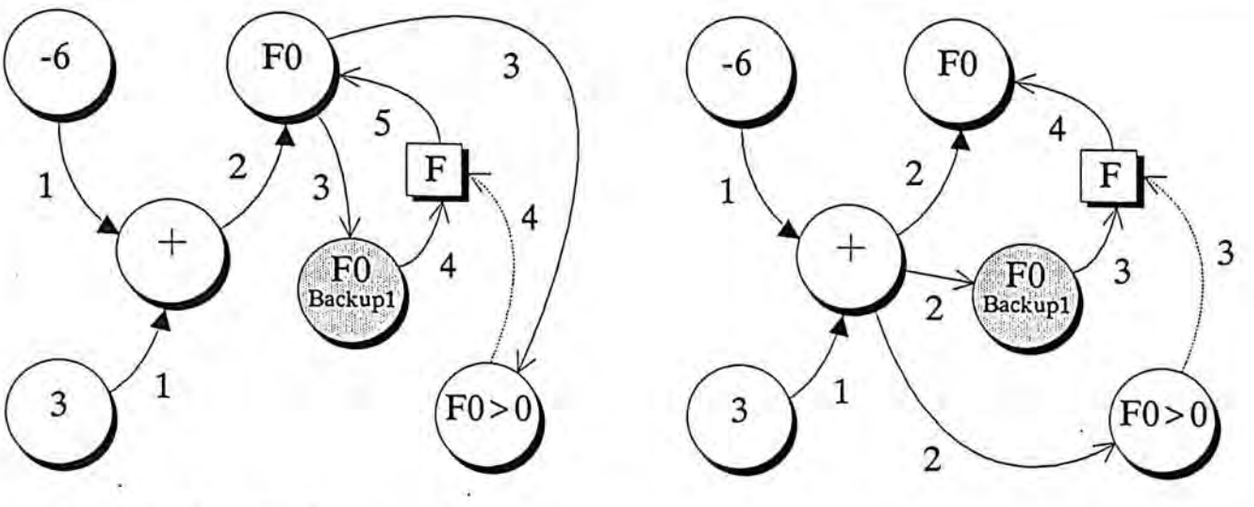


Figure 3.5. An example to illustrate the use of the T-Operator and the F-Operator

With the F-Operator, the handling of the BZ instruction is represented neatly as depicted by the procedure graph in figure 3.6. The shaded node "F0 Backup1" denotes a "private" data area for storing a backup copy of the value of the register F0 just before the execution of the BZ instruction. Doubtless, it should be the output of the ADD instruction, as shown in figure 3.6.

By "private", we mean that it is not user-addressable as an ordinary storage location. Hardware components such as the program counter which are supposed not to be accessed directly by the programmer explicitly will be represented by shaded nodes when they are strictly required to appear in a procedure graph. In fact, the branch backup and repair mechanism should be totally transparent to the programmer.

Branch Prediction results in conditional execution being continued from the MOV instruction labelled by @10. In figure 3.7, the final procedure graph is drawn.



(a) Procedure graph representing the first 3 instructions of figure 3.5(a)

(b) Transformed procedure graph of the one shown in (a) by applying the SP on :
arcs '+'→'F0' and 'F0'→'F0>0'
arcs '+'→'F0' and 'F0'→'F0 Backup1'

Figure 3.6. The use of the F-Operator to represent the BZ instruction in figure 3.5(a)

Several points should merit further consideration. By applying an SSC, the result of the ADD instruction appearing before the BZ instruction will not affect the register F0. In other words, the SSC has been applied across a basic block[2] boundary. What's so significant is that we are no longer constrained to do local optimization only. Global optimization of limited scope (bounded by the maximum level of branch nesting allowed) can be achieved. Its correctness is guaranteed by the branch backup and repair mechanism.

[2] A *Basic Block* refers to a straight-line sequence of instructions that does not contain a branch or a branch target within except at the end. A basic block is credited for its *'single-entry-single-exit'* characteristics - the flow of control enters only at the beginning and leaves at the end without halt or possibility of branching except at the end.

In spite of the above, the ADD instruction will still affect the backup copy for register F0 to prepare for the possible exceptional event. At the same time, its result will determine the outcome of the BZ instruction also.

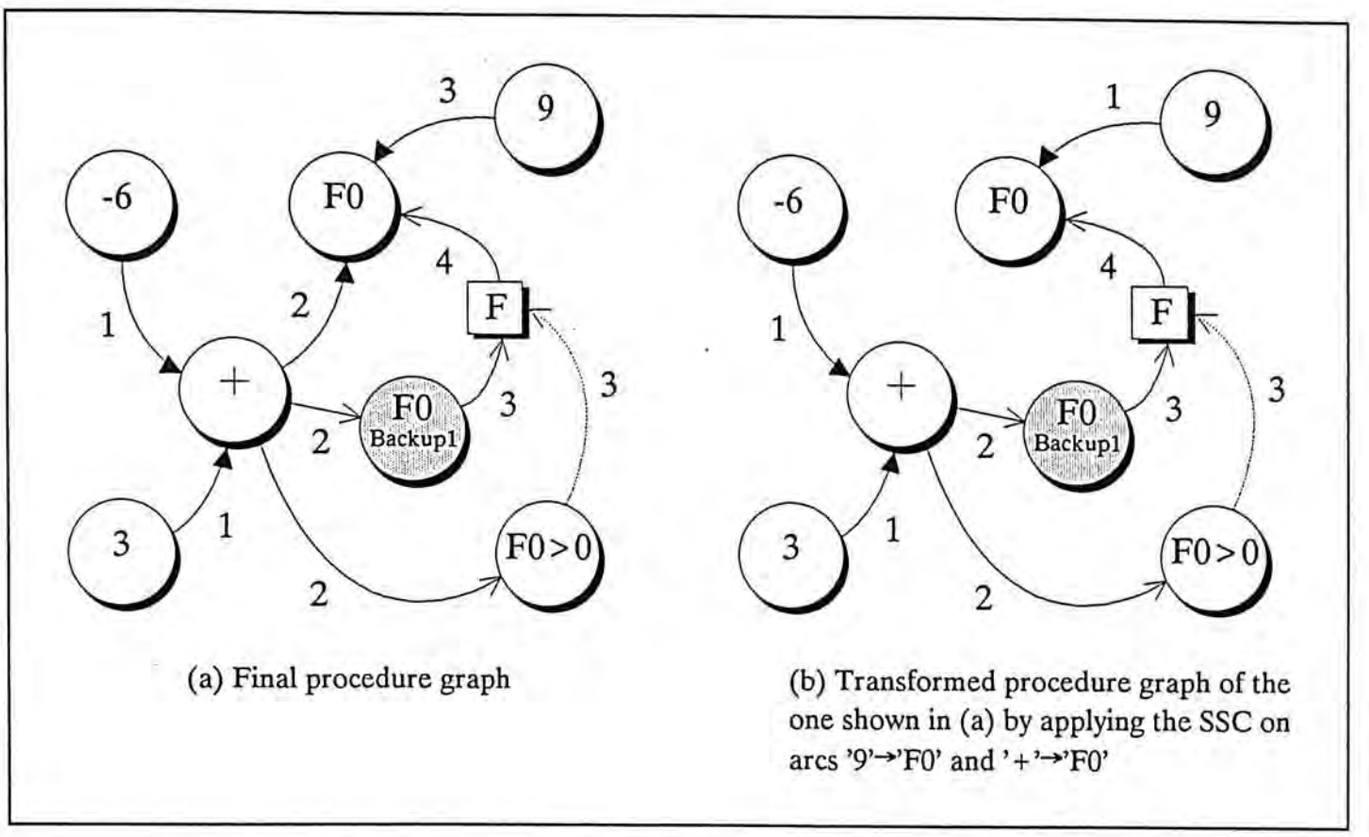


Figure 3.7. The final procedure graph for the code sequence in figure 3.5(a)

Although we still have to perform the ADD operation because of the procedural dependency, the MOV instruction can be executed immediately. More important, later independent instructions can be expedited as early as possible. One may argue that this represents a waste of computation effort in some sense. However, with the replication of hardware has brought about.

To conclude this section, we consider the limitations of the model. First, only the backup for a single register has been provided. In a real situation when the whole set of registers have to be restored upon a faulty branch, we may need to create multiple backup nodes. Alternatively, we can choose to have one single backup node to stand for

the whole set of machine registers. In addition, the problem will be further complicated if nested branching is allowed, as is common for loops.

3.4 Procedure Graph Representation For a Stack

In this section, we consider an application of the T-Operator in developing a procedure graph representation of a stack. Basically, it is a shift-register simulation scheme. We assume that the stack is top-pointing and is of limited size. As an example, we consider a stack of a total capacity of 3 elements, as represented by the three shaded nodes 'Stack Item 1', 'Stack Item 2' and 'Stack Item 3' in figure 3.8.

In our model, only two operations are allowed - popping and pushing. The access of the stack obeys the First-In-First-Out rule strictly. In other words, no topping of stack is allowed, and at any time, only the element at the top of the stack, if any, is referable.

Whether an operation is applicable will be governed by various constraints. Their implementation constitutes a critical part of our model. On the one hand, one should not be allowed to pop from an empty stack. On the other hand, a full stack should prohibit further pushing of new data, or inconsistency and loss of data may result.

With reference to figure 3.8, the node labelled by 'Size' keeps track of the number of elements remained in the stack, which is incremented by each push and decremented by each pop. Again, the use of a shaded node signifies that the storage location is not user-addressable.

Should the stack be empty when we try to pop out the top-most element from it, the operation will be declared as invalid and the value of 'Size' should not be decremented (modified). Doubtless, there is a dependency involved and we should first make sure that the current value of 'Size' is greater than zero before we can decrement

it by one. With reference to figure 3.8, the output of the operator 'Size>0' (which is TRUE if 'Size' is greater than zero and FALSE otherwise) is fed as the control input for the adder. If it turns out to be false in fact, the value of 'Size' is protected from being modified by inhibiting the firing of the adder. Similar mechanism has been adopted to check for the validity of a push operation. Now the output of the operator 'Size<3' acts as the control signal.

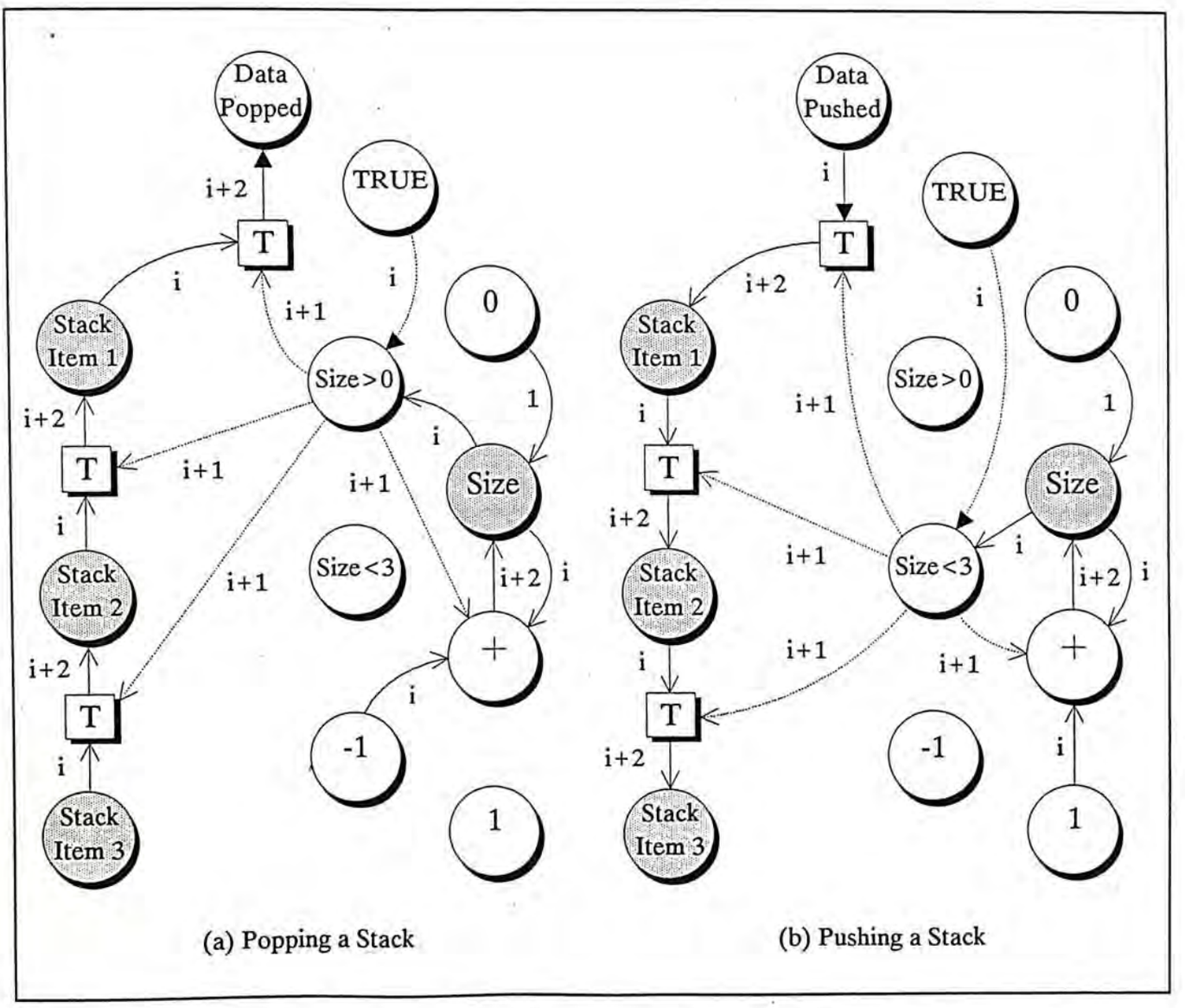


Figure 3.8. Basic operations of a stack implemented with the help of the T-Operator

As a final comment, we consider some limitations of this model. First, only finite stack can be considered and the generalization would involve the creation of many stack

element nodes. The resulting procedure graph will unavoidably be complicated. On the other hand, we can see that there is a fan-out problem at the operator nodes 'Size>0' and 'Size<3' which exists even if the stack is constrained to hold a few elements only. A possible remedy is by relaying the control signal at the element nodes. For example, the control signal from 'Size>0' or 'Size<3' can be passed from Stack1 to Stack2 and then to Stack3. Doubtless, the propagation delay resulted will lengthen the access time for pushing and popping the stack. When the maximum capacity of the stack increases, the situation will further deteriorate and finally, the representation scheme may simply turn out to be infeasible at all.

3.5 Vector Forwarding

The discussion of procedure graph transformations and forwarding should not be restricted to scalar data only. Vector operands should be given equal weight. And we believe that greater performance gain can be realized than scalar forwarding, with only minor extensions to the original theory. Vector forwarding is also discussed in [Chen92].

3.5.1 An Example of Vector Chaining in Cray-1

Consider the example in figure 3.9, the VRs denote vector registers (in Cray-1, a vector register is implemented as an array-like structure with 64 elements each). Pseudo-time labels are now abstracted by algorithms. The result of such an algorithm is a list of scalar time labels describing successive data transfers along the edge. Distinction should be made between an algorithm which generates a list of labels with multiple labels explicitly specified as a list. While the latter represents a time-shared situation and successive data items transferred are loosely-related, the former does not. In fact, it is both a consequence and intention that a situation in which successive data items of an array are involved is to be exhibited.

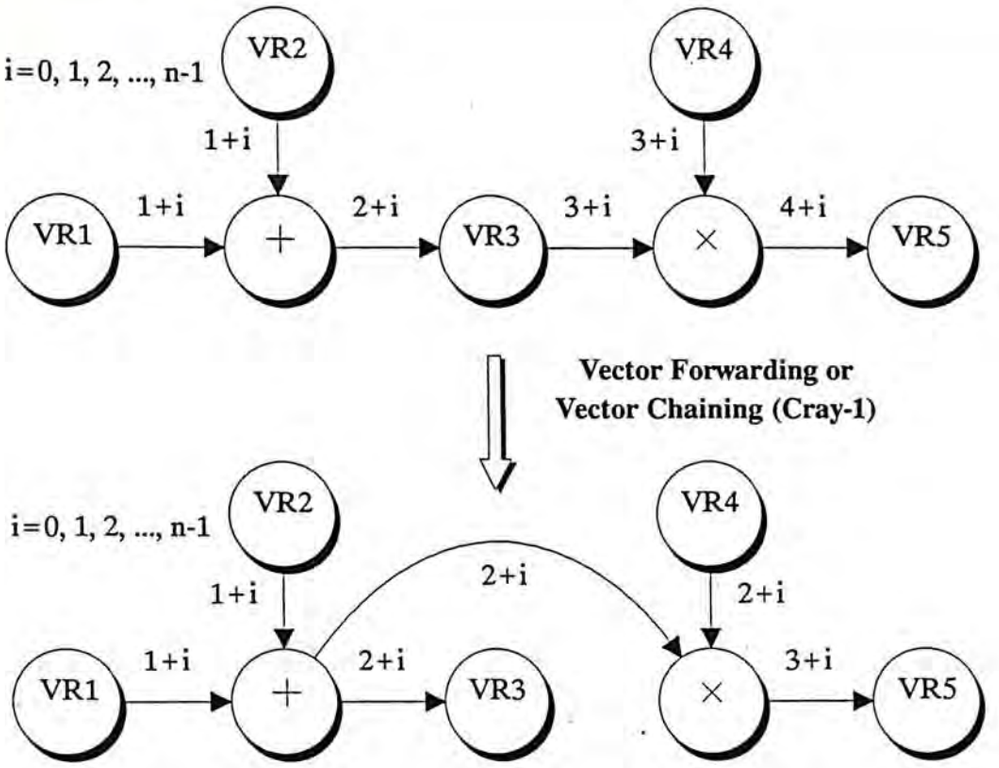


Figure 3.9. An example of Vector Forwarding or Vector Chaining in Cray-1

Let's return to the discussion on figure 3.9. The sequence of results by summing successive corresponding elements from VR1 and VR2 are now forwarded to the multiplier directly, thus bypassing the vector register VR3. Effectively, a Vector Serial-to-Parallel Transformation is achieved. Similar techniques had been successfully implemented in the Cray-1 Machine [Hwang&Briggs84]. The shortening of total duration as shown should not be the whole story. More important, in the worst case, the first multiplication just cannot proceed until each of the 64 additions performed and every vector component of VR3 contains valid data. In such circumstances, the performance gain will be profound and significant.

3.5.2 Vector SP, PS and SSC

Extending the three classical internal forwarding rules to include vector operands gives rise to the situations in figures 3.10(a), (b) and (c). Several points should be made regarding the algorithmic time labels involved. Concerning Vector SP, in the original

graph, the vector time labels T2 and T1 should produce the same number of scalar time labels on enumeration[3]. Besides, if the intended purpose is to make a copy of VR3 into VR1 and VR2 respectively, then each scalar label produced by T1 should be strictly smaller than the respective one produced by T2. In addition, after transformation, T' should preserve the same objective. In other words, each component of T' should be greater than or equal to the corresponding scalar label produced by T1. A natural decision is to make T'=T1.

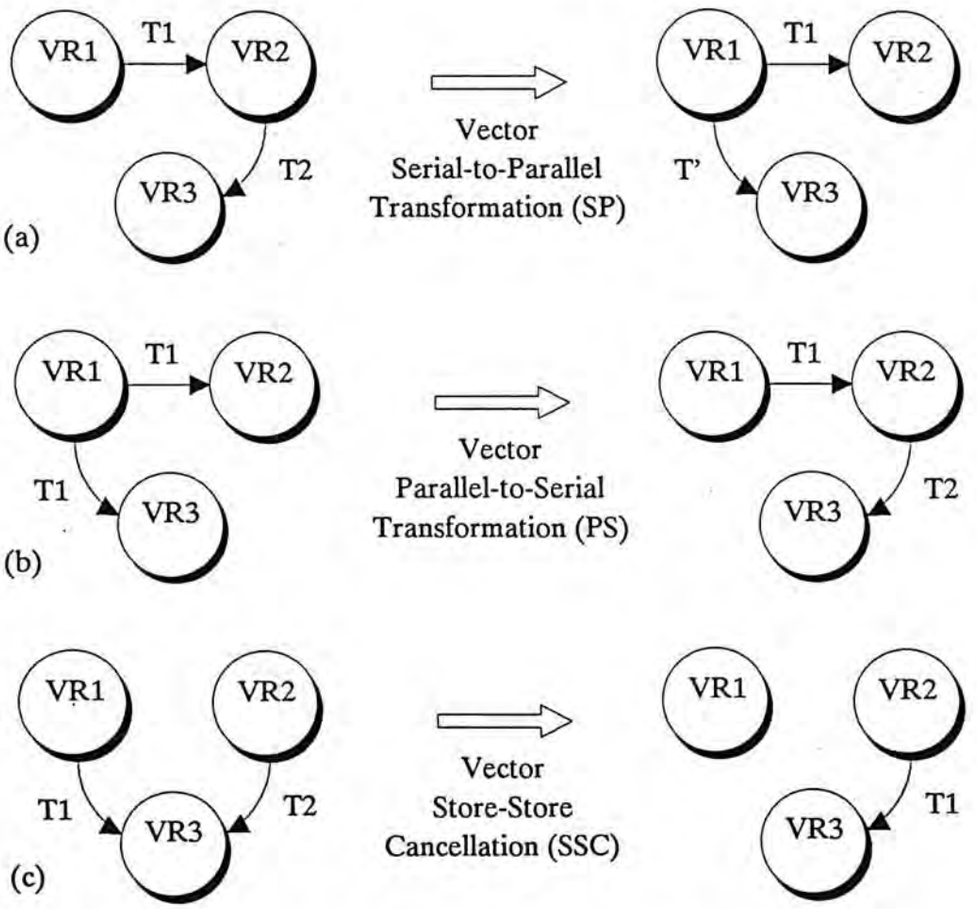


Figure 3.10. Vector Procedure Graph Transformations/Forwardings

Similar arguments apply in the case exhibited in figure 3.10(b) for a Vector PS. However, things become complicated when dealing with Vector SSC. To correctly override the earlier sequence of transfers from VR1 to VR3, one should first make sure

[3] The Cray Machine requires that if T1 is represented by t+i (where i=0, ..., 63), T2 will be denoted by t+i+64. In other words, there is (at least) a 64-step delay.

that each scalar time label produced by T2 should be strictly greater than the respective one suggested by T1. Otherwise, inconsistencies arise and we would be forced to manipulate selected components of the vector only, rather than treating it as a whole.

3.5.3 A Note Concerning the Use of Algorithmic Time Labels

Let's recall the two new perspectives on precedence time labels : for denoting successive data transfers of consecutive components of a vector over time, and the repeated transfers from a scalar location/data over time. The distinction between them is especially concerned when interaction of scalar and vector should be considered. While the former encodes information of both time and (storage) space, the other concerns events in successive time intervals only.

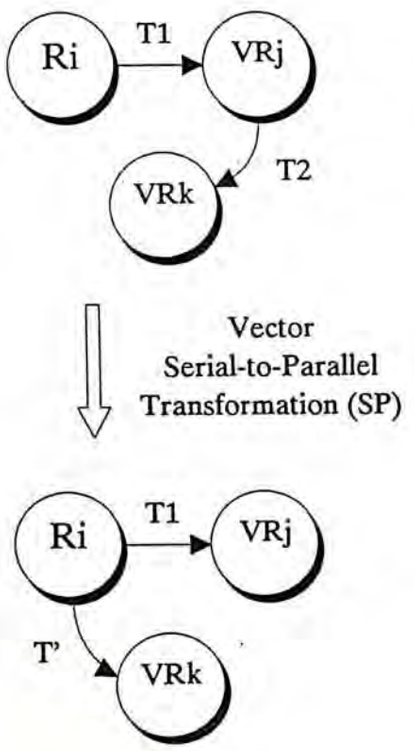


Figure 3.11.
A Vector SP involving scalars and vectors

Consider the case in figure 3.11 where VRj and VRk are vector registers and Ri denotes a scalar register. Familiar enough, the vector time labels T1 and T2 should give rise to the same number of elements. More important, although they both enumerate a list of time labels, their types are different. With the (SP) transformation, not only is the duration shortened and direction of data transfers changed, the type of the time label is also switched, To make things simple, we choose not to invent two different types of algorithmic time labels. Rather, the context will provide the necessary information. When the source of an arc is a scalar location (e.g. a scalar register), the algorithmic time labels should imply a sequence of successive data transfers from the same scalar location.

3.5.4 Further Consideration of Vector Forwarding

There is something interesting concerning the interaction of a scalar and a vector that worth going into. Looking back to the example in figure. With reference to the Cray Machine, there is a constraint governing the use of the time labels T1 and T2, namely, the first transfer of data from (the first element of) VRj to (the first element of) VRk cannot proceed until the last store from Ri into (the last component of) VRj completes. Figure 3.12 below is an illustration.

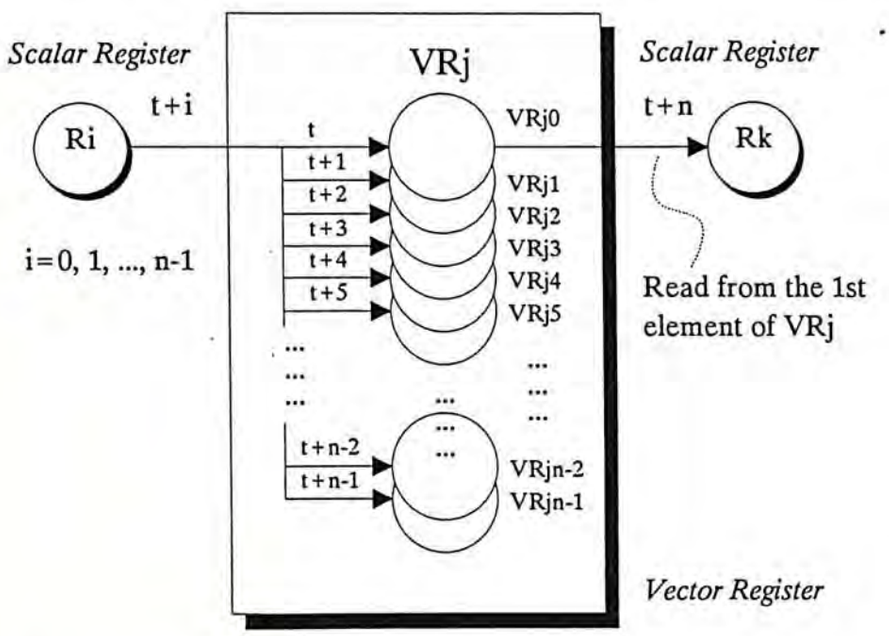


Figure 3.12. The n-step delay incurred in transferring data from a scalar to a vector (Cray-1)

Doubtless, the n-step delay (if the number of elements of each vector register is n) is very undesirable and performance suffers much. As proposed by T. C. Chen, we can construct a more efficient design for a vector register using multiplexers and demultiplexers, as shown in figure 3.13.

The innovation has saved us from the n-step delay. Its advantages can be made explicit by considering the example exhibited in figure 3.14. The effect achieved by the "fanout" resembles that of a "broadcast". The direct consequence is that the content of

an element of an vector register will be available right after it is loaded, and we no longer have to wait until the total array of register cells are all manipulated. An application involving the initialization of an entire vector array of registers would find this design especially appealing.

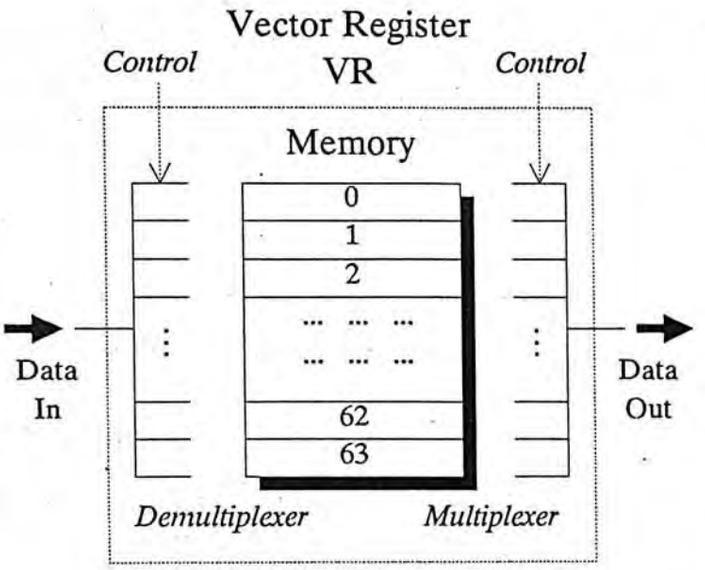


Figure 3.13. Implementing a vector register using multiplexers and demultiplexers

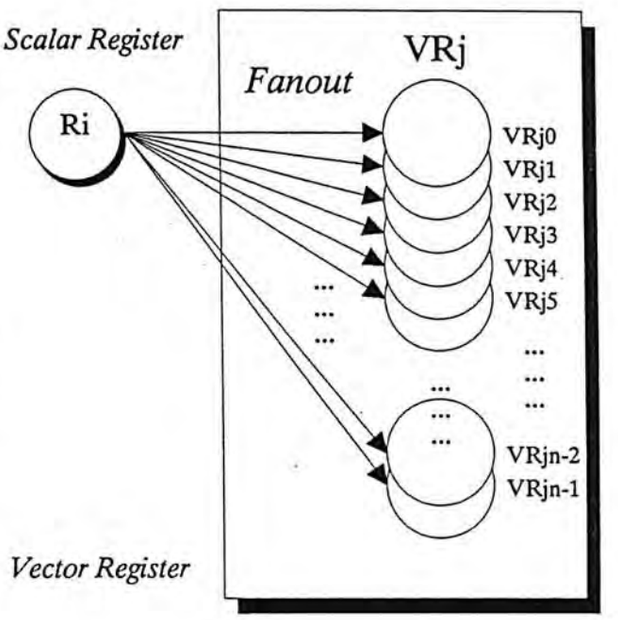


Figure 3.14. More efficient data transfers from a scalar register to a vector register

Chapter 4 Hardware Realization Of Procedure Graph Optimizations

4.1 Node-Oriented Versus Arc-Oriented Representation

To discuss the structural properties of a procedure graph, the basic elements are its nodes and arcs. One can choose to represent a procedure graph by its set of nodes or set of arcs. Each carries its own advantages and disadvantages.

First, with a Node-Oriented Representation, we enumerate for each node the set of incoming arcs and/or the set of outgoing arcs. When the information of an arc is stored at its source node (respectively its sink node), only the sink node (respectively the source node) has to be recorded. The resulting forward-threaded (respectively backward-threaded) list, when traversed, will reveal the downstream (respectively upstream) arc-information only. Alternatively, we can also choose to store the information of each arc at both its source and sink, resulting in a doubly-threaded list. But then a single arc will have to be mentioned twice and we have to record at the source node its sink, and at the sink node its source. The redundancy concerned is obvious, especially when arcs have multiple pseudo-time labels. In spite of this, the inter-relationship between arcs will be better preserved. This facilitates easy analysis and manipulation of procedure graphs.

On the other hand, in an Arc-Oriented Representation, the set of arcs are enumerated, specifying for each its source node and sink node. In terms of storage requirement, both approaches are comparable. However, inter-arc information is not directly available for arc-oriented schemes and some kind of search mechanism should adopted. Take for instance, it is very difficult, if not impossible, to locate the set of all predecessors and successors of an arc. This requires a global examination of the whole set of arcs.

As an example, we recall the simulation program of procedure graphs discussed in chapter 2. Basically, an Arc-Oriented scheme is adopted with procedure graphs represented using Prolog's items/structures. Thus the procedure graph in figure 4.1(a) will be manifested by the following "database" of Prolog structures :

```
arc(N1,[1],N2).
arc(N2,[2],N3).
```

where each "arc(Source,Pseudo_Time_Label,Sink)" represents a data transfer arc leading from "Source" node to "Sink" node annotated with "Pseudo_Time_Label". To account for multiple pseudo-time labels on arcs, a Prolog list is used for "Pseudo_Time_Label". As shown, no inter-arc information is stored and we rely on the "instantiation" mechanism of the Prolog programming language to achieve arc manipulations and procedure graph transformations.

To conclude, we can see that there is a tradeoff and between the two extremes of Arc-Oriented Representation and Node-Oriented Representation, we have a wide range of choices.

4.2 Backward Pointers Versus Forward Pointers

The potential redundancy involved in Node-Oriented Representation can be avoided by storing the information of an arc only once, either at its source node or its sink node, but not both, manifesting itself as a forward pointer or backward pointer respectively. The best choice should be made by evaluating such criteria as the amount of redundancy involved and the exact problem to be modeled by the graph (which should be able to reveal what kind of graph manipulations will be required frequently).

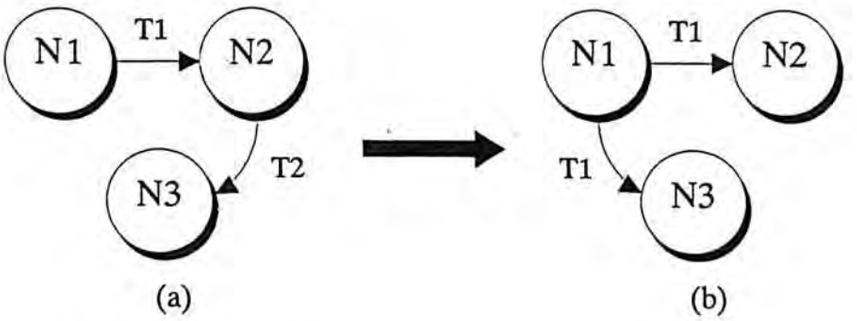


Figure 4.1. A Serial-to-Parallel Transformation (T2 > T1)

Consider the procedure graph depicted in figure 4.1(a). If arc-information are stored at the sources, we have the following :

Forward Pointers

At node N1

"There is an arc leading to node N2 at time T1"

At node N2

"There is an arc leading to node N3 at time T2"

At node N3

< None >

On the other hand, if arc-information are stored at the sink nodes, we obtain the representation scheme below :

Backward Pointers

At node N1

<None>

At node N2

"There is an arc leading from node N1 at time T1"

At node N3

"There is an arc leading from node N2 at time T2"

In terms of storage-efficiency, both schemes consume the same amount of space. But when access-efficiency is concerned, the advantages of backward pointers become

apparent. Two observations should help to justify our argument. First, the applications of various procedure graph transformations involve the manipulations of data transfer arcs. The use of backward pointers facilitates these operations. As an example, in order to achieve the Serial-to-Parallel Transformation from figure 4.1(a) to 4.1(b), we simply have to redefine the source node of the backward pointer stored at N3, resulting in the following representation :

Backward Pointers

At node N1

<None>

At node N2

"There is an arc leading from node N1 at time T1"

At node N3

"There is an arc leading from node N1 at time T1"

However, if arc-information are stored at the source nodes, the same graph transformation will require two operations then. The forward pointer at node N2 should be deleted and a new forward pointer has to be added to node N1, obtaining the following :

Forward Pointers

At node N1

"There is an arc leading to node N2 at time T1"
"There is an arc leading to node N3 at time T1"

At node N2

<Node>

At node N3

< None>

More importantly, there are difficulties in actual implementation. When each data transfer arc is mentioned only at its either end, the access efficiency will be

sacrificed as a trade-off for saving storage. In particular, if backward pointers are used, only immediately upstream information can be easily accessible. The opposite is true for forward-pointer representation schemes. While downstream arc-information can be directly obtained by following the end-to-end pointers in the forward direction, given a node N, it is generally very difficult to access those arcs with N as their sinks. Some kind of global view of the procedure graph should be provided.

We encounter a similar difficulty in our example if forward pointers are used. The data transfer arc N2→N3 alone does not suffice to locate the node N1 and the arc leading from N1 to N2. In other words, another access mechanism should be provided in order to realize the Serial-to-Parallel Transformation concerned.

Still another inefficiency incurred by the use of forward pointers has been revealed in this example. While the mapping from sink nodes to source nodes is generally unique (an exception being the case of arithmetic operators/nodes), the inverse usually does not hold. The fact that a single node can have more than one outgoing data transfer arc each leading to a different sink. as in the situation depicted in figure 4.1(b), implies that multiple 'slots' may have to be associated with the source node to enumerate the information of each of these arcs.

In other words, we need to maintain a list of addresses for delivering each single piece of data to multiple destinations. Doubtless, the overhead involved in its allocation, deallocation and manipulation will complicate the hardware much. Even worse, as the number of destinations is theoretically unbounded, any number of list entries may become insufficient in some case.

On the other hand, such problem can be avoided in backward-pointer representation schemes. As dictated by the semantics of procedure graphs, two data transfer arcs leading to the same sink node should be annotated with different pseudo time labels. Equivalently, simultaneous writes to the same storage location must be

strictly prohibited[1]. Therefore, a single slot, though with limitations (to be explained later) will suffice.

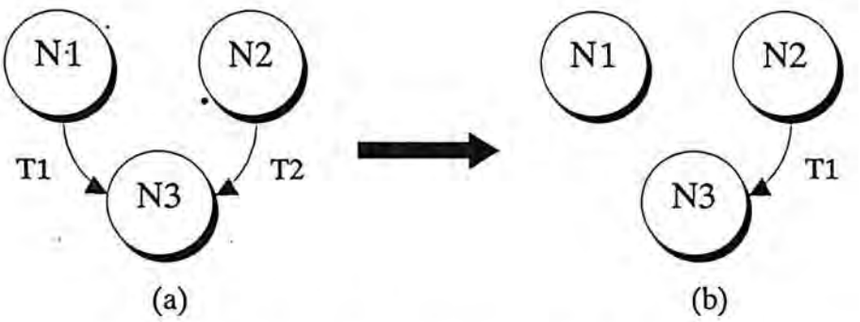


Figure 4.2. A Store-Store Cancellation (T2 > T1)

More importantly, by restricting that only a single (incoming) arc's information can be associated with a node, Store-Store Cancellations are implied naturally. With reference to figure 4.2(a), when the data transfer arc 'N2→N3' is encountered, an attempt to store it at its sink node N3 will reveal that the single slot there has been occupied by the arc 'N1→N3' already. A replacement of it by the arc 'N2→N3' effectively applies a Store-Store Cancellation, resulting in the procedure graph depicted in figure 4.2(b)[2]. But if forward pointers are used, the information of these two arcs will be stored separately at their respective source nodes N1 and N2. As a result, Store-Store Cancellations become difficult to apply.

4.3 Backward Pointers As Hardware Tags

The first successful implementation of procedure graph optimizations was the floating-point execution unit of the IBM 360/91 [Tomasulo67]. An algorithm under execution is

[1] In computer operations, while simultaneous output or fanout is common and being a major source of parallelism to promote efficiency, simultaneous input (of signals) to a single storage node is usually an error.

[2] In actual implementation, the assignment "N1→N3" should precede "N2→N3" in the sequential instruction stream. As a result, we don't have to check for the constraint T2 > T1 before applying the Store-Store Cancellation. Similar argument applies for the Serial-to-Parallel Transformation depicted in figure 4.1.

represented as a procedure graph at the hardware level using backward pointers which manifest themselves as hardware tags. The joint effort of tagging, forwarding and optimizing procedure graph transformations have achieved a very high performance level of the machine.

A simple tag is associated with each prospective recipient of data (e.g. register), which corresponds to a storage node in a procedure graph. With reference to figure 4.3, if the Valid Bit V is 1, the corresponding Data field stores the most updated (or valid) content of N. On the other hand, if V=0, an instruction in execution is going to define its final content and the Tag field identifies this source (see [Chen80]).

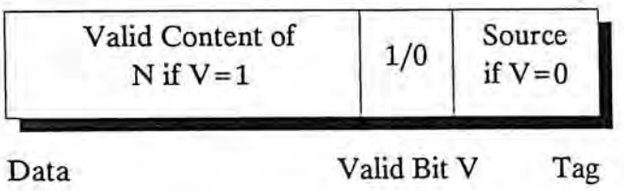


Figure 4.3. A Hardware Tag

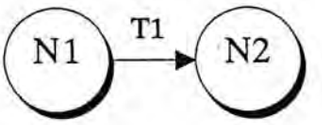
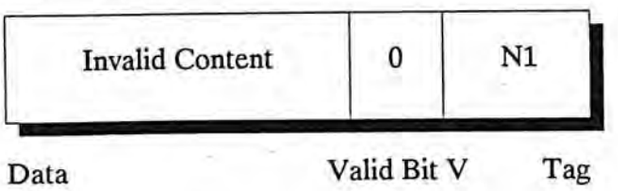


Figure 4.4. A Data Transfer Arc

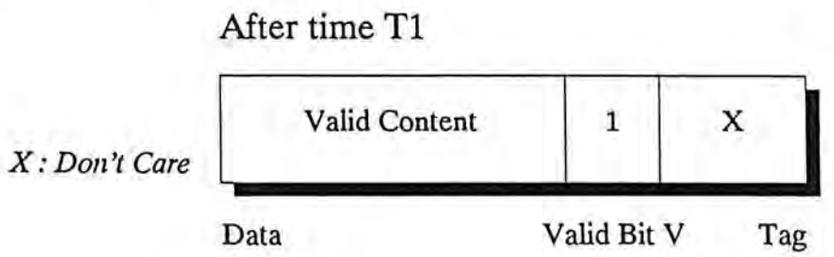
For example, to represent the data transfer arc depicted in figure 4.4, the following tag will be kept by the node N2 which denotes an "intention to receive" from N1 :

Before time T1



A clear distinction between the data transfer arc and the hardware tag should be made here. While an event (a data transfer from "N1" to "N2") is manifested by the former, the hardware tag describes the state of the sink node N2. The actual occurrence of the event at time T1 corresponds to the availability/arrival of the data (from N1)

which triggers the change of state of the sink node. Therefore, after time T1, the hardware tag at the node N2 will be modified as :



Just as a note, any consistent identification for N1 can be used as the source tag of N2. In fact, it needs not be unique as well. In some designs (e.g. [Sohi&Vajapeyam87]), a "tag pool" is used where tags as names are allocated and reused dynamically. At one time, "Name1" may be associated with the node N1 and referred in the Tag field of N2. When the data is ready, "Name1" accompanies it on the Common Data Bus. N2 with its matching tag will then make a copy of the data. When it is finished, "Name1" will be released, which can now be bound to a different node, say N3.

Alternatively, we can also associate a fixed "name" for each prospective producer of data which serves as its unique identification to the recipients. This is precisely the approach adopted by the IBM 360/91.

The need to distribute a single piece of data to multiple destinations has been urged by the repeated applications of Serial-to-Parallel Transformations which turn relay data transfers into parallel "broadcasting". As pointed out earlier when we discussed the disadvantages of using forward pointers, the solution by keeping a list to enumerate each destination address is on the one hand clumsy and inefficient, and more importantly, as the number of destinations can be theoretically unbound, any length of such a list may appear to be insufficient in some case.

Therefore, a different approach was adopted in the IBM 360/91. For each data to distribute, instead of remembering "who wants it", it requires every prospective destination to know "what it wants". A Common Data Bus is implemented which is capable of reaching each possible destination in the CPU in the same cycle. A destination waiting for a piece of data will keep a unique identification of it (which is the "name" or the tag that we have been talking of earlier). When the data finally becomes ready and appears in the Common Data Bus accompanied with its identification, destinations with a matching tag will gate in a copy individually.

A major drawback of the Common Data Bus is that only one piece of data can be distributed in each cycle. We will come to this issue again in later chapters where we evaluate the performance bottleneck incurred by the Common Data Bus.

4.4 Pointer Algebra

Procedure graph transformations are achieved via the manipulations of the hardware tags - Pointer Algebra.

4.4.1 Serial-to-Parallel Transformations

With reference to figure 4.5(a), the Tag field of N2 identifies the source node N1 as the originator of its valid content[3].

[3] Two interpretations are possible here. On the one hand, we may have encountered a load instruction LD N2,N1 (N2←N1) with N1 being the memory location/address of interest. On the other hand, N1 can also represent a functional unit to execute the arithmetic instruction Op N2,N (N2←N2 Op N). The data transfer arc N1→N2 will thus denote the output of the result. In either case, N2 represents the sink register and the data from N1 is not immediately available.

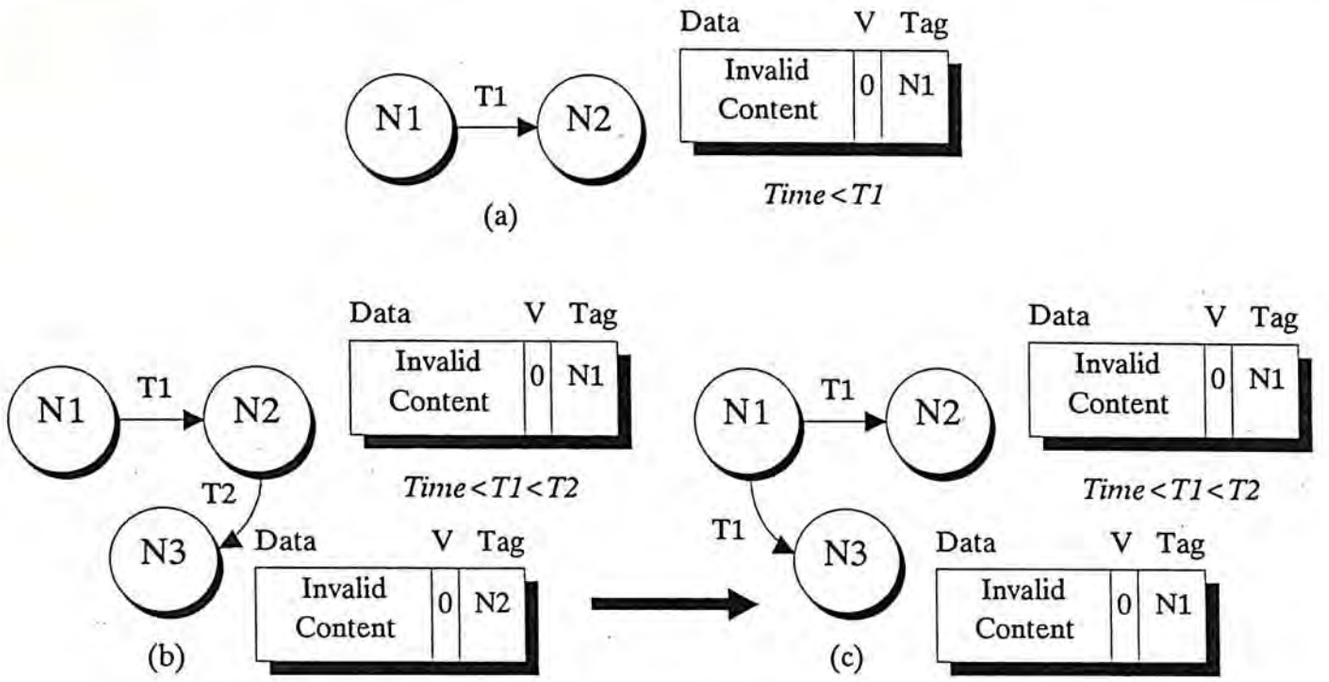


Figure 4.5. Achieving a Serial-to-Parallel Transformation (b→c) by tag copying

As the data transfer arc N2→N3 is considered (see figure 4.5b), the hardware tag at N2 is examined. A value of 0 in the Valid Bit V indicates that the content of N2 is not valid yet. As a result, the Tag field of N2 is copied to the hardware tag of N3 with its Valid Bit reset to 0 at the same time, resulting in the situation shown in figure 4.5(c)[4]. Equivalently, a Serial-to-Parallel Transformation is achieved. When the data from N1 is finally ready and appears in the Common Data Bus with its identifying tag, each of N2 and N3 will receive a copy (of the data) separately.

4.4.2 Store-Store Cancellations

As shown in figure 4.6(a), suppose the data from N1 is not ready yet. Consequently, the Tag field of N3 will be set to locate N1 as the source of its most updated content and the Valid Bit is reset accordingly. When the data transfer arc N2→N3 is encountered (see figure 4.6b), we simply overwrite the original content of the Tag field of N3 by the new source N2. In this way, when the data from N1 later appears in the Common Data Bus,

[4] On the other hand, if the Valid Bit of N2 is 1, meaning that its content (or the data from N1) has been ready, the Data field will be copied instead and the Valid Bit of N3 will be set to 1.

it won't modify the content of the node N3. As shown, the fact that only a single tag (or backward pointer) can be associated with a node implies Store-Store Cancellations naturally (see figure 4.6c).

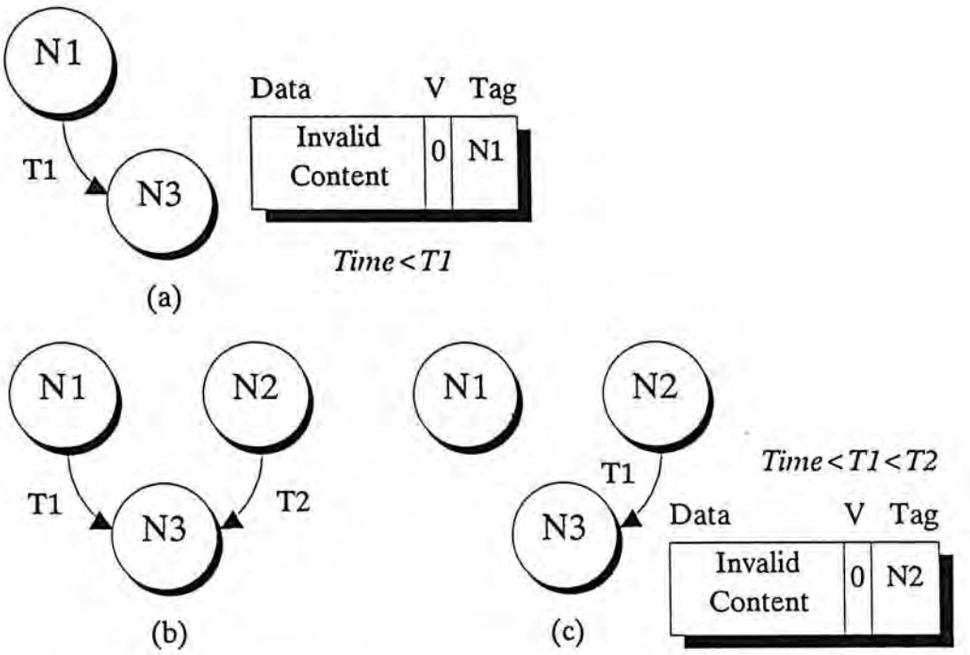


Figure 4.6. Achieving a Store-Store Cancellation (b→c) by tag overwriting (T2 > T1)

4.4.3 Parallel-to-Serial Transformations

When data transfer arcs are represented by backward pointers (or hardware tags), Parallel-to-Serial Transformations become difficult to apply. Consider the case illustrated in figure 4.7.

To achieve the transformation depicted, the architecture should have a mechanism to reach the node N2 (as well as the arc N1→N2) given the data transfer arc N1→N3 (or the corresponding tag at N3) and the node N1. But in any pure backward-pointer representation scheme, the information of the arcs N1→N3 and N1→N2 will be separately stored at the node N3 and N2 respectively (as tags). In other words, it is very difficult, if not possible, to obtain the arc-information downstream directly, inhibiting easy applications of Parallel-to-Serial Transformations.

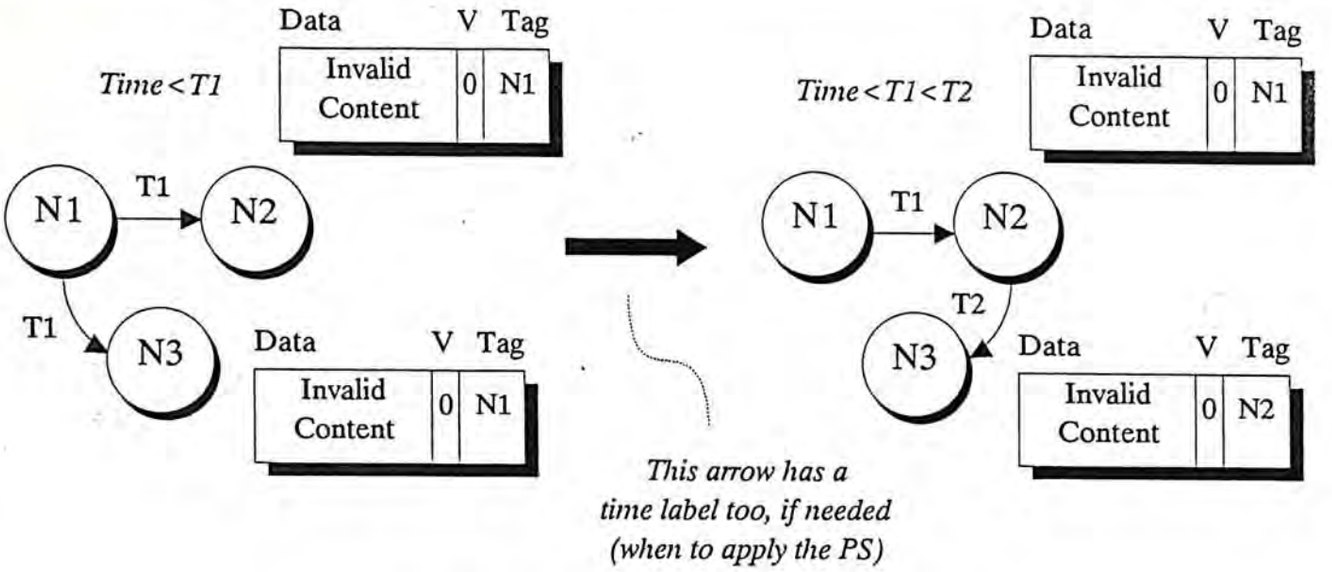


Figure 4.7. A Parallel-to-Serial Transformation

4.5 Drawbacks Of Using Backward Pointers

As a conclusion to our discussion, we summarize the advantages and disadvantages of backwards and forward pointers in tables 4.1 and 4.2 below.

Table 4.1. Achieving Transformations

	Backward Pointers	Forward Pointers
SP	*Good*	*Bad*
PS	*Bad*	*Good*
SSC	*Good*	*Bad*

Table 4.2. Handling Fan-in and Fan-out

	Backward Pointers	Forward Pointers	Frequency
Fan-out	*Good*	*Bad*	*High*
Fan-in	*Bad*	*Good*	*Very low*

To conclude, backward pointers with all the advantages mentioned above should be preferred. However, some points merit further discussion. First, as pointed out earlier, when backward pointers are used for representing data transfers, it will be very difficult to obtain the arc-information downstream, that is, the successor(s) of a data arc. We have just seen how this inadequacy has inhibited the applications of Parallel-to-Serial Transformations. But whether this is strictly necessary and when it will be required are questions worthy of further consideration. The doubly-threaded list representation scheme adopted in the CDC6600 computer [Stone87] (strictly speaking, it is an arc-

oriented approach), though can help to provide more global view of the procedure graph or algorithm under execution, incur significant redundancies and access overhead.

On the other hand, while graph transformations (and optimizations) can be achieved effectively and efficiently via manipulating simple hardware tags, the fact that no more than one single arc's information can be stored at a node has constrained us to do real-time[5] optimization only as we have no way to look into events that will happen later. The scope of applications of equivalent graph transformations is dictated by the linear sequencing of the instructions. And one is forced to examine the computation from the very beginning only. Even worse, when the data at the source node has been valid already, no forwarding can be done. Doubtless, this lack of predictive or lookahead capability will limit both the scope and effectiveness of the optimization achieved.

4.6 Multiple Tags

The use of procedure graphs gives rise to a new computation model. To solve a problem, we think of an algorithm and then code it. After compilation, the machine code will be executed by our target machine.

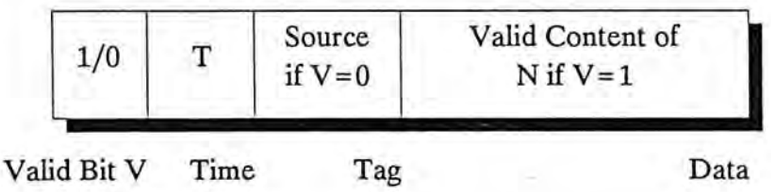
The basic idea of our model is to represent the total computation by a global procedure graph with pseudo time labels. To optimize operations, we extract/map a subgraph of it, identify an equivalent graph, and then perform the transformation (if the "economic" factors are favoured). The resulting subgraph will be stitched back into the original graph. Having preserved the causality of operations and data transfers, the transformed graph will represent the same computation as before.

[5] Strictly speaking, it is not real-time. Rather, it should be "just before real-time". The optimization is achieved after the assignment but before the arrival of data.

In fact, we are looking for equivalences of entire sub-algorithms. Every time we focus on a subset of the total computation (which is not constrained to be the very beginning of the algorithm only). By replacing it with a more efficient equivalence, the overall performance can be promoted. The fact that the computation performed will not be affected is guaranteed by the rule of associativity which holds among the different sub-algorithms of a program.

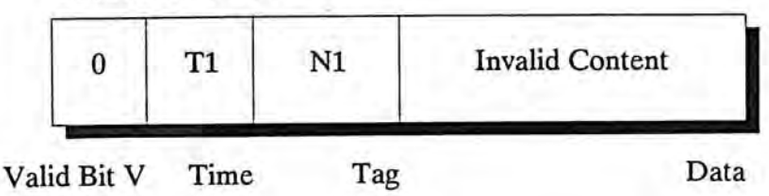
Unfortunately, this cannot be achieved in a simple tagged architecture since we have no information concerning events which will happen later, that is, instructions which are pending for execution.

In view of this, the Multitag Architecture is invented. We start by modifying the original simple tag. An element of "Time" is added, leading to the following Timed Tag :

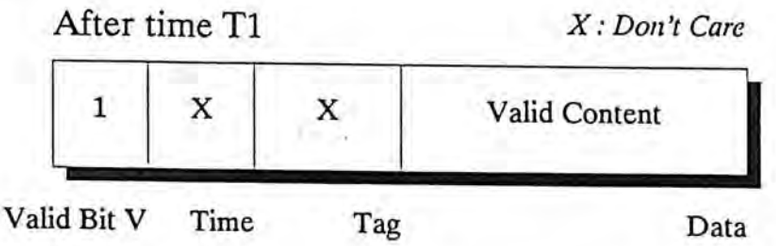


If the content of N is not ready as indicated by its Valid Bit V (reset to 0), the timed tag tells us that it will be supplied from the source identified by the Tag field at time T. Thus to represent the data transfer arc in figure 4.4, before time T1, the following timed tag will be associated with the sink node N2 :

Before time T1



Upon the receipt of the data from N1 at time T1, the tag at the node N2 will be modified as :



More importantly, to account for the multiple data transfer arcs that may be incident onto a single node, each prospective data sink (storage location or functional unit) can now have multiple backward pointers or tags instead of just one (for arithmetic node, separate tags will be associated with each input reservation station). And the computation to be performed will be manifested by them. We believe that this is a true image of a procedure graph at the hardware level. Illustrated in figure 4.8 is the procedure graph representation of the Gaussian Elimination inner loop involving one element only with the set of tags shown (to make things clear, we choose to omit the right operands of + and × which are supposed to travel along the dashed arcs).

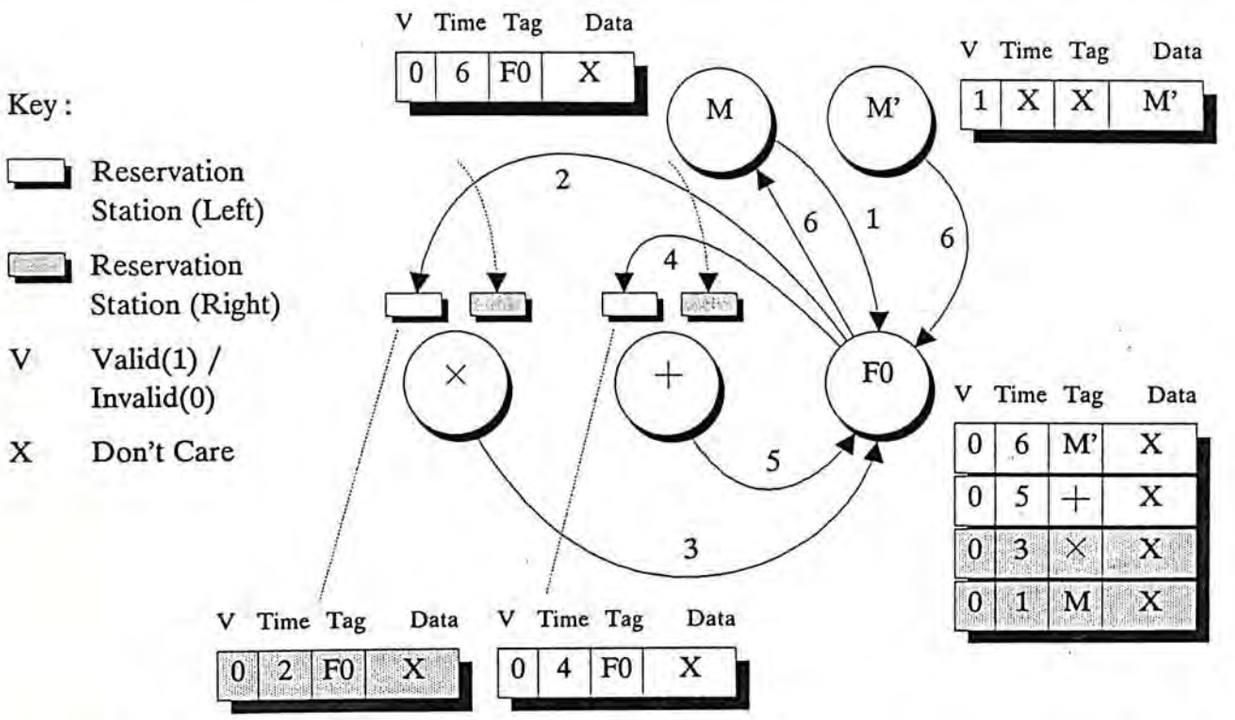


Figure 4.8. The original procedure graph representing the Gaussian Elimination inner loop

(Only one element is considered here. Multiple timed tag(s) is/are associated with each node or reservation station. The shaded tags will be selected for consideration when a dL transformation [Chen91] is applied to derive the graph in figure 4.9)

The execution of transformation between equivalent graphs are now achieved via manipulating the timed tags. To optimize a subgraph/sub-algorithm, we select those tags of interest. For example, we may focus on those tags with the Time field equal to 3,4 or 5 (effectively, we consider those arcs in the corresponding procedure graph labelled by 3,4 or 5), and look for equivalent transformation(s) involving them only.

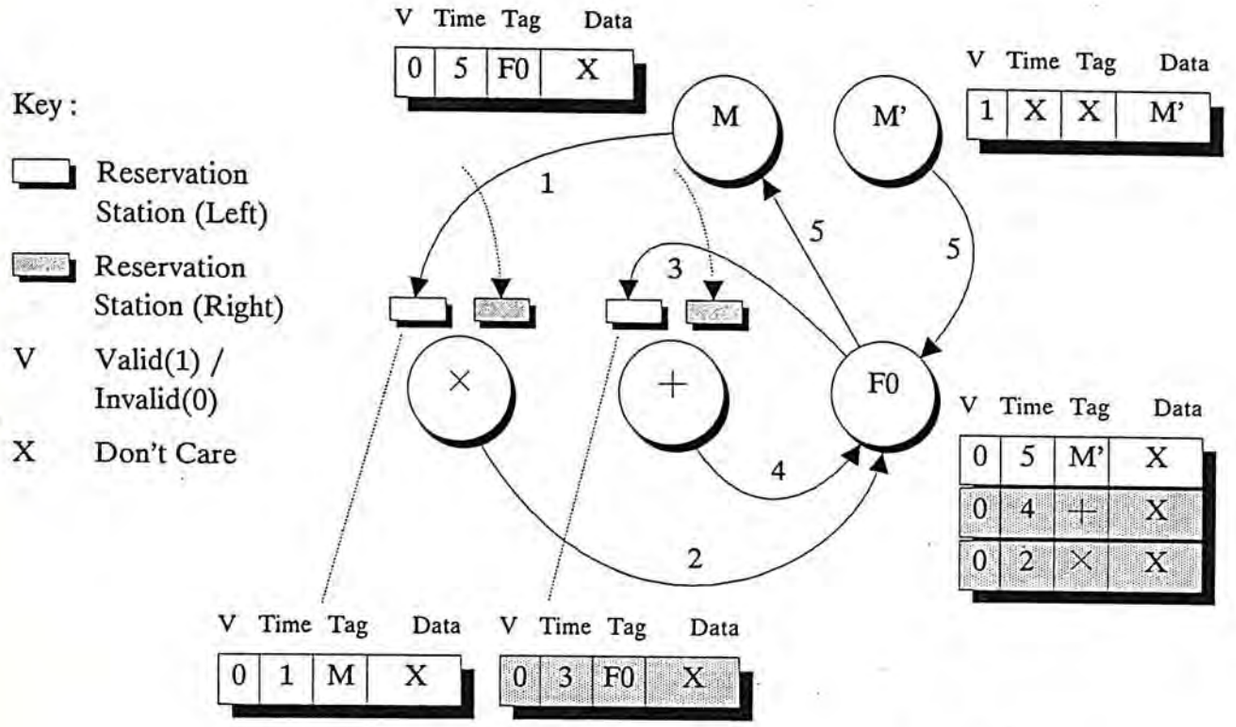


Figure 4.9. The procedure graph derived by applying the dL Transformation dL(M F0 × F0) to the original graph shown in figure 4.8

The actual transformation between equivalences is effected by updating the tags involved. Surrounding tags may also be affected when normalization or re-labelling is needed. In figures 4.9, 4.10 and 4.11, the example involving the optimization of the Gaussian elimination inner loop is repeated from [Chen91], with emphasis on the changes of tags. The major advantage of the Multitag Architecture is that we no longer need to start from the very beginning every time. Arbitrary sub-algorithm can be optimized. That is what the procedure graph theory specifies. The innovative use of pseudo time labels in representing the causality of computation has on the one hand

given clear-cut boundaries of sub-algorithms, and more importantly, facilitated the identification of equivalence and the transformation between them.

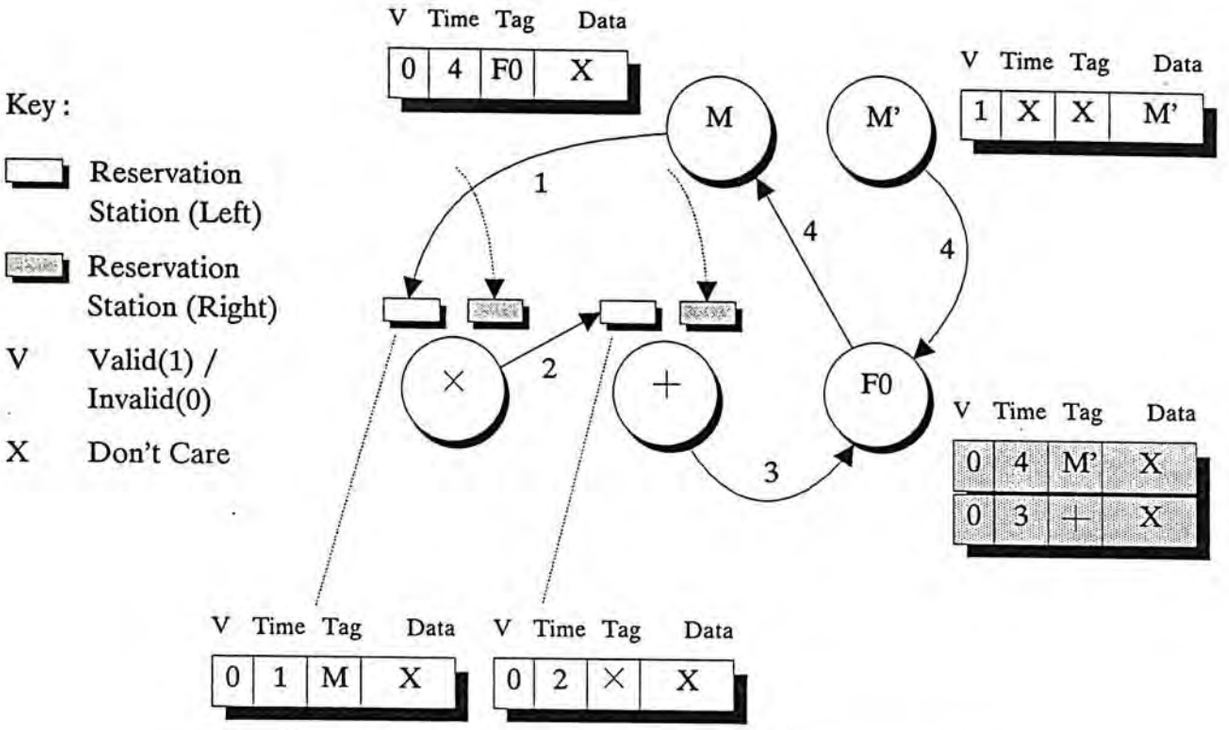


Figure 4.10. The procedure graph obtained by applying the dL Transformation dL(× F0 + F0) to the graph shown in figure 4.9

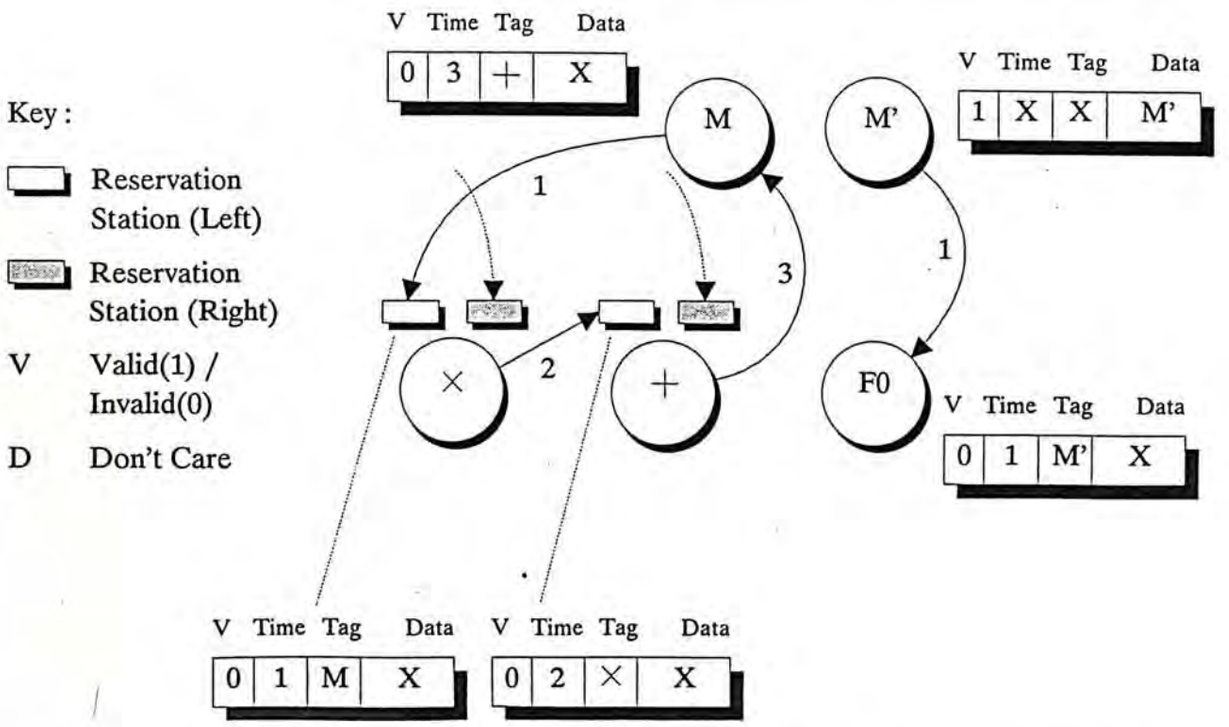


Figure 4.11. The optimized procedure graph for the Gaussian Elimination inner loop

(From figure 4.10 to figure 4.11, the dL Transformation dL(+ F0 M ... F0) is used)

With multiple tags, predictive optimization as allowed by the availability of tags can now be achieved. We are in fact working towards a hardware implementation of simple compiler optimization techniques. But to have a successful implementation of this Multitag Architecture, we still have some difficulties. First, as the graph (or computation) becomes complicated, the multiple tags will pile up at the same time. This introduces a representation problem for the tags. In addition, as we may have to deal with arbitrary subset of computation (subgraph). each tag should be individually and efficiently referable. This suggests that a mechanism has to be adopted for managing them globally. Finally, the generation of the set of tags from an algorithm (or program) is a really interesting problem in compilation which merits further discussions.

Chapter 5 A Backward-Pointer Representation Scheme : The T-Architecture

5.1 The T-Architecture

The first successful implementation of procedure graph optimizations was the floating-point execution unit of the IBM 360/91 [Tomasulo67]. An algorithm under execution is represented as a procedure graph at the hardware level using backward pointers which manifest themselves as hardware tags (see chapter 4). The joint effort of tagging, forwarding and optimizing procedure graph transformations has achieved a very high performance level of the machine.

As an attempt to further exploit the advantages of procedure graph optimizations, we have designed the T-Architecture (meaning Tagged Architecture), as shown in figure 5.1. Important enhancements have been added to the original model of the IBM 360/91. In search of high performance, the T-Architecture represents an integration of various techniques - procedure graph optimizations, superscalar techniques, speculative execution, tagging and forwarding.

As in the IBM 360/91, two procedure graph transformation rules have been implemented in the T-Architecture, namely, Serial-to-Parallel Transformation (SP) and Store-Store Cancellation (SSC)[1]. The policy is - "perform SP and SSC as far as possible". The rationale is straight-forward. On the one hand, the total duration can be shortened by changing relay data transfers into parallel broadcasts via SP Transformations. On the other hand, the benefits of deleting dummy operations are also obvious.

The delay in data transfer and operation brought about by Parallel-to-Serial Transformation (PS) makes it unfavorable for implementation. More importantly, as

[1] Refer to section 4.4 for a discussion of the underlying mechanisms involved in achieving an SP and an SSC.

revealed by our discussions in chapter 4, the use of backward pointers inhibits the easy application of PS.

The memory system has been revised. An Updating Buffer is adopted (see figure 5.1), facilitating memory data forwarding. Superscalar techniques are introduced. By duplicating the instruction fetch unit and the decoder, multiple instructions can now be considered at the same time. The increase in the scope to optimize should promote the effectiveness achieved.

In addition, aiming at overriding procedural dependencies and improving fetch efficiency, branch prediction and speculative execution have been adopted. Procedure graph transformations can now be applied across basic block boundaries. Restricted global optimization as allowed by the maximum branch level will bring along more significant speedup. All these characterize the T-Architecture as a high performance machine relying mainly on hardware optimizations (or dynamic instruction scheduling). In the succeeding sections, we will consider each of them in detail, with emphasis on the use of backward pointers.

Simulation results reveal that a performance level of over 0.96 instructions per cycle can be attained by the T-Architecture (the fetch size being equal to 1), meaning a 96% of the maximum efficiency. More importantly, we have achieved it without the help of software optimization techniques. This demonstrates the usefulness of the procedure graph theory and backward-pointer representation schemes.

5.2 Local Addressing Space Within the CPU

In the T-Architecture, we have devised a separate addressing scheme which is local to the CPU only and is disjoint from the memory address space. With reference to figure 5.1, a unique Local Address has been assigned to each prospective originator of data which identifies itself to the prospective recipients. Each destination waiting for a data

will keep in the Tag field of its hardware tag the Local Address of the data source. It is this Local Address that will accompany the data in the Common Data Bus when the latter is finally available.

Two advantages are obvious with this Local Addressing space. First, it saves the overhead involved in the allocation and deallocation of tags when they can be dynamically bound to different originators. Besides, by buffering memory operations in reservation stations also (i.e. providing a level of indirection using the Store Reservation Stations and the Load Reservation Stations), only a few bits (instead of a long memory address, say 32 bits) will suffice for each Local Address. This highly facilitates applying the concepts of tagging and forwarding to memory data also.

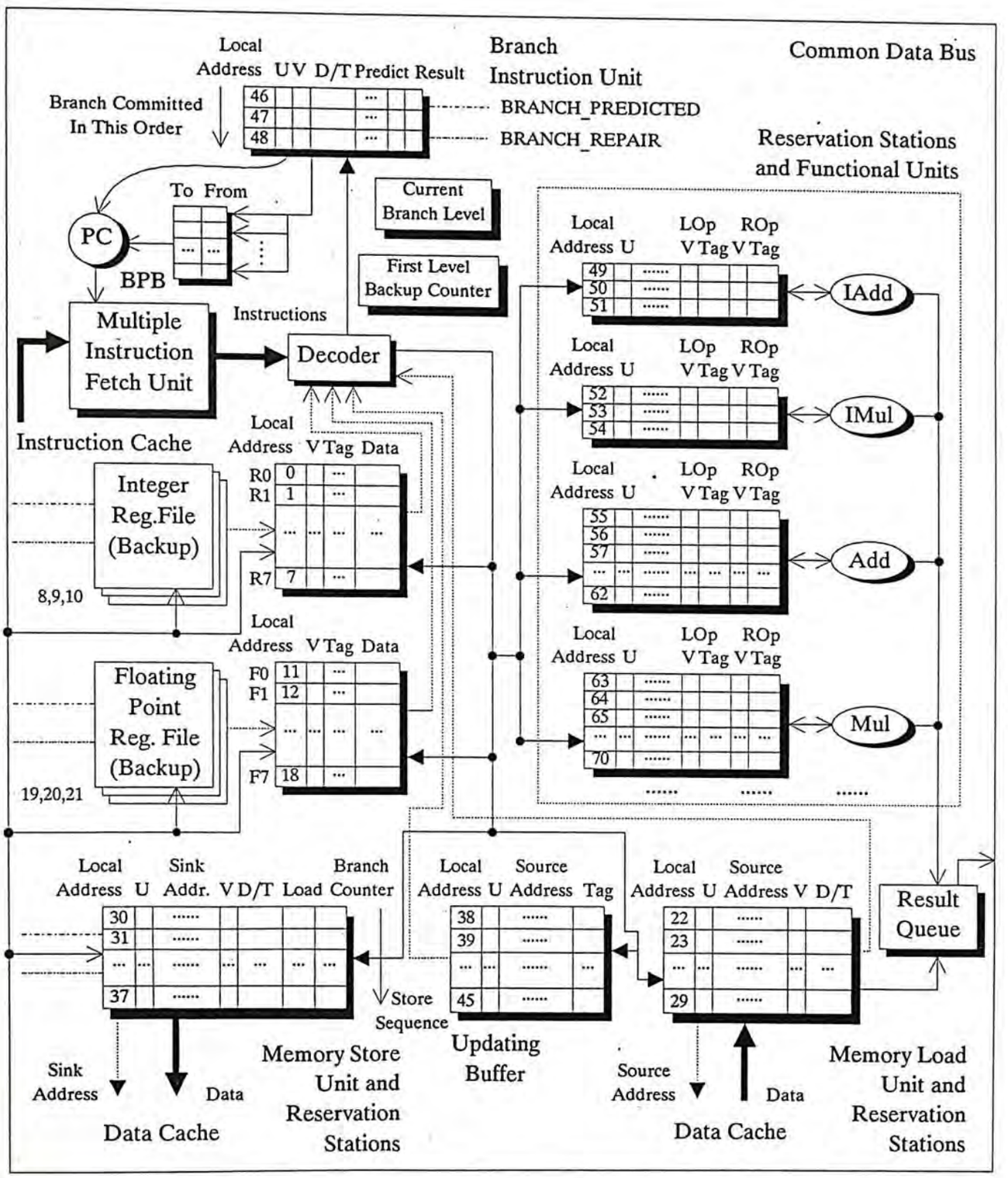
5.3 Why Reservation Stations

The use of reservation stations was first proposed in the design of the IBM 360/91 as an attempt to resolve the resource conflicts for functional units. In terms of the procedure graph theory, the adoption of reservation stations splits a single operator node into several temporary storage nodes, each being capable of buffering one pending operation.

According to M. Johnson [Johnson91], the reservation stations effectively function as an 'instruction window' between the decoder and the functional units. Instructions upon decoding are placed in the instruction window. Multiple instructions are examined together. Succeeding operations which are ready to be fired are allowed to override the ordering implied by the original sequencing of instructions. As a result, they will not be unnecessarily blocked (because of resource conflicts or unnecessary precedence constraints created by the use of a sequential programming language)[2]. Out-of-order issue is achieved and the executions of instructions are expedited as early as

[2] This issue is more significant for superscalar designs because the stalling of a 'wider pipeline' will result in a greater performance loss.

possible.



Key :

BPB Branch Predicting Buffer
Issue Operations (Issue Bus)
Tag or Data
Tag and Data
BRANCH_PREDICTED
BRANCH_REPAIR
U Used Bit
V Valid Bit
D/T Data or Tag

Figure 5.1. The T-Architecture

On the other hand, this instruction window is a distributed one. An operation in a reservation station remains dormant until all of its source operands become valid. Every time when a functional unit becomes free, it is arbitrated among the ready-to-go operations (of the same type). Therefore, control can be distributed and multi-threaded. Operations, once decoded, reside in reservation stations where they can proceed autonomously and in an asynchronous manner, realizing the maximum degree of overlapped and out-of-order execution as allowed by data dependency and availability of computing resources.

To illustrate the usefulness of reservation stations, consider the following code sequence :

```
LD      F0,A[1]
MUL     F0,@3
ST      A[1],F0
LD      F0,A[2]
MUL     F0,@3
ST      A[2],F0
LD      F0,A[3]
MUL     F0,@3
ST      A[3],F0
ADD     F1,@1
```

Suppose the content of the register F1 is currently valid. Thus the ADD instruction is ready to be fired immediately. A system with only one floating-point multiplier and no reservation station would be forced to execute the algorithm as in figure 5.2(a). Upon applying procedure graph transformations, we arrive at the procedure graph depicted in figure 5.2(b). Although significant improvement has been realized, the three multiplications still have to queue for the single multiplier to become free. The ADD instruction, though is ready to execute already, turns out to be unnecessarily held.

To achieve further speedup, reservation stations are adopted for the floating-point multiply unit, resulting in the situation in figure 5.3 (now, the floating-point multiply unit can be perceived as having a multiplier plus a certain number of reservation stations). Although the three multiplications still have to queue for the single multiplier, the Add is no longer blocked from issue and can proceed immediately.

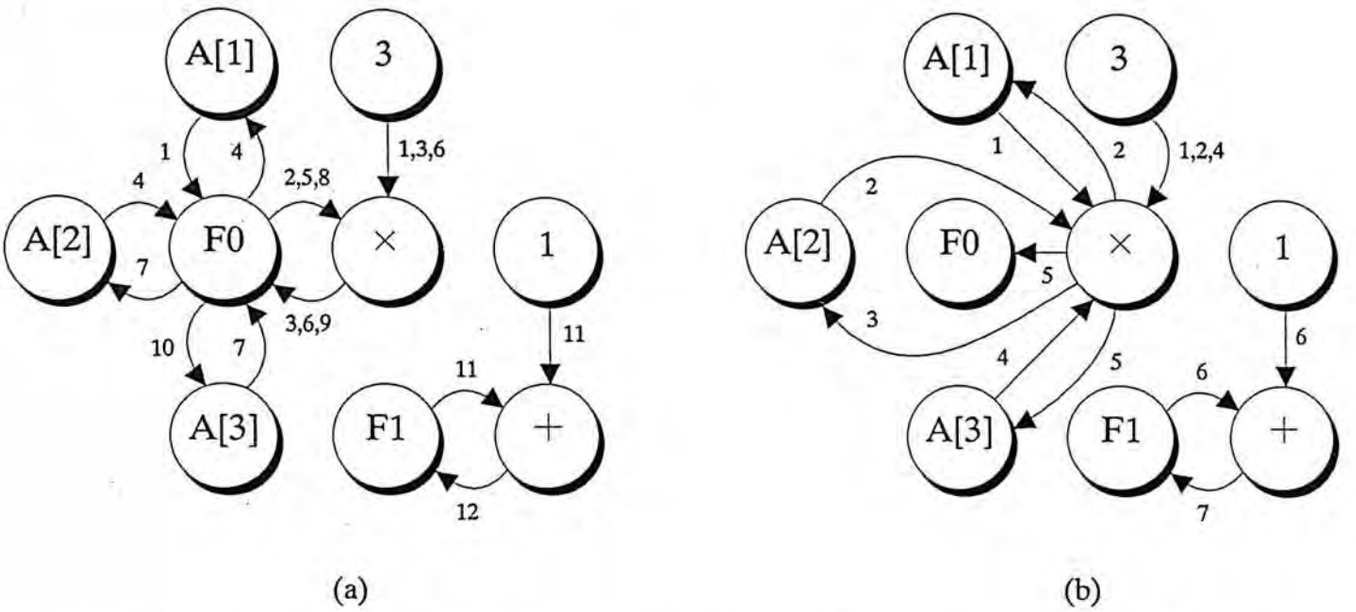


Figure 5.2. Without reservation stations, decoder is stalled until the multiplier becomes free

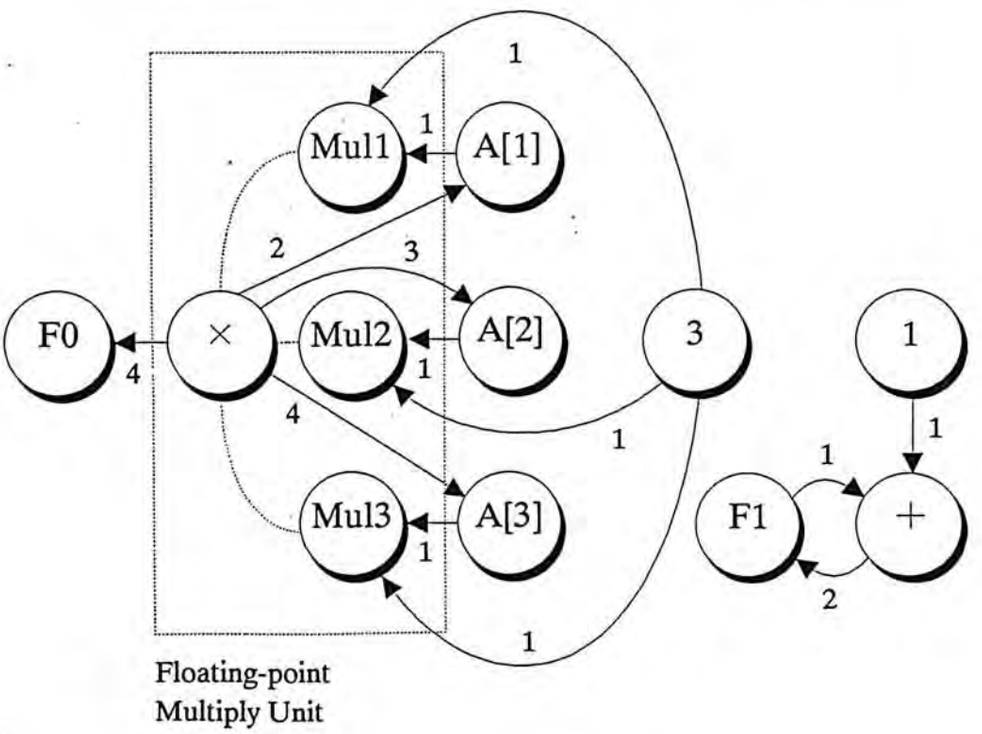


Figure 5.3. With reservation stations, the ADD is no longer blocked from issue

(× : Multiplier; Mul1, Mul2 and Mul3 : Multiply Reservation Stations)

More importantly, if enough floating-point multipliers are available, the three multiplications can be executed in parallel, as in figure 5.4. Without buffering, we simply are not aware that the three multiplications are in fact independent because the instruction fetch unit and/or the decoder, having recognized that the multiplier is busy, would have been stalled when the second multiplication is encountered. In other words, the performance bottleneck incurred by the resource conflict has been overridden. Maximal parallelism is attained with performance only constrained by Read-After-Write dependency (or true data dependency).

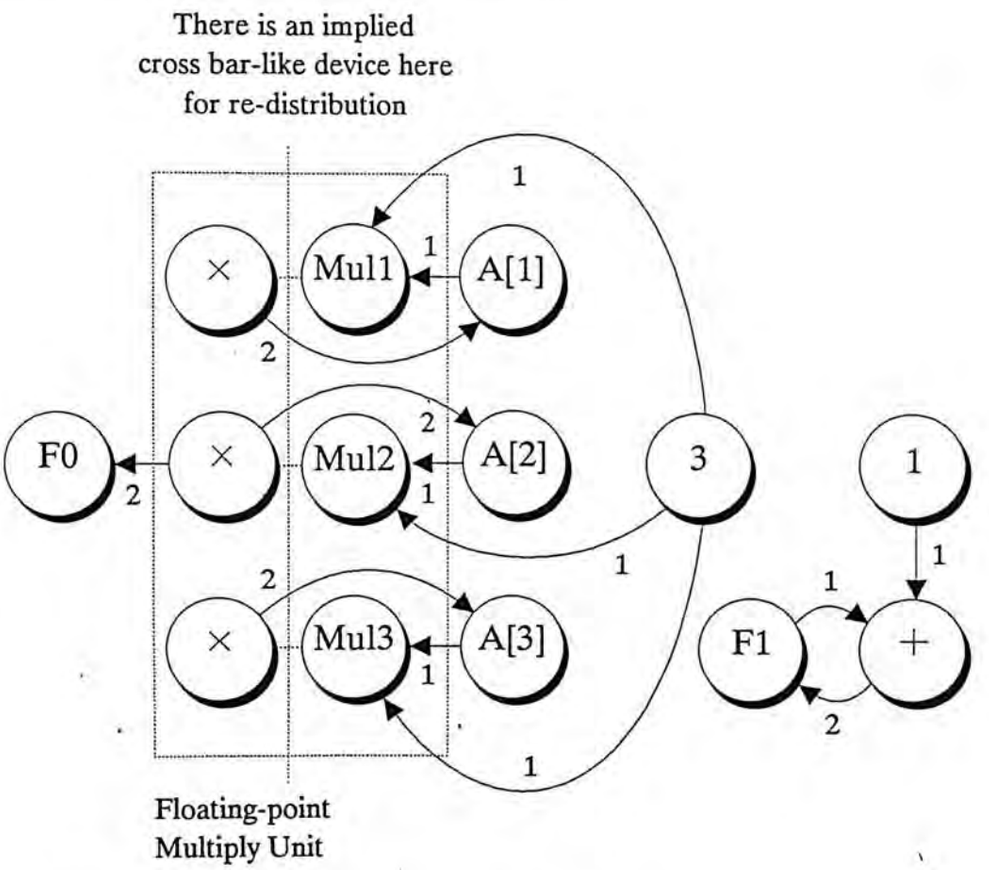


Figure 5.4. If 3 multipliers are adopted, the 3 multiplications can proceed in parallel

(× : Multiplier; Mul1, Mul2 and Mul3 : Multiply Reservation Stations)

As a final comment, we can see that it is the reservation station instead of the functional unit that identifies a particular computation result.

5.4 Memory Data Forwarding

In the classical von Neumann model, we have a Memory Address Register (MAR) and a Memory Data Register (MDR). To handle a load operation, the address of the memory location of interest is placed in the MAR and upon the completion of the memory access, the MDR will contain the data read. When a store is required, the data to store and the destination address should be placed in the MAR and the MDR respectively and the store operation initiated.

In any case, only one memory access can be considered with one MAR and one MDR. Even if a separate pair of registers, known as Store Address Register (SAR) and Store Data Register (SDR), are dedicated for memory store operations exclusively, no more than two memory accesses can be outstanding at the same time.

While an increase in the number of read ports and/or write ports can expand the bandwidth of a memory system, the fact that only one memory access can be buffered (or two accesses in case SAR and SDR are present) will simply leave most of these resources idle. What we require is an increase in the number of MARs and MDRs at the same time. This need is further urged in a superscalar architecture in which the key to performance is multiple fetch as well as multiple and out-of-order execution with the correct causality preserved. By buffering extra memory accesses which may be either waiting for free read/write ports or are simply not ready for initiation because of data dependency (e.g. the data to store is not valid yet), succeeding instructions will not be blocked (which may be ready for execution immediately). This is true even if the bandwidth of the memory can sustain only one access at the same time.

In the T-Architecture, a number of Load Reservation Stations and Store Reservation Stations are adopted (see figure 5.5). Intuitively, each Load Reservation Station functions as a MAR-MDR pair while each Store Reservation Station serves as a

SAR-SDR pair. As a result, multiple memory accesses can be outstanding at the same time.

(a) A Load Reservation Station

Local Address	U	Source Address	V	D/T

(b) A Store Reservation Station

Local Address	U	Sink Address	V	D/T	Load	Branch Counter

Key :
U : Used Bit
V : Valid Bit
D/T : Data or Tag

Figure 5.5. The structure of a Load Reservation Station and a Store Reservation Station

Each Load Reservation Station (as shown in figure 5.5a) is capable of buffering one load instruction/operation. The Source Address locates the data to be loaded and the unique Load Address of each Load Reservation Station identifies the data read to its prospective recipients. When a load operation is completed, the data read together with the Local Address of its originating Load Reservation Station are put on the Common Data Bus. Those with a matching Tag will gate in a copy of the data.

Similarly, each Store Reservation Station can be reserved for one store operation exclusively. With reference to figure 5.5(b), if the data to be stored at the Sink Address is not ready yet, the Valid Bit V will be reset and the D/T field identifies the data it wants.

5.4.1 The Updating Buffer

The consequence of having multiple MARs and MDRs is that more than one memory access (either executing or pending for initiation) may be addressing the same location simultaneously. Some kind of 'directory' is needed. With reference to figure 5.1, the memory system of our design has adopted an Updating Buffer to navigate all memory

accesses and resolve potential conflicts. More importantly, by providing an extra level of indirection, procedure graph transformations can be extended to memory data also. The advantage of having such a "central management" is that all accesses to the Load and Store Reservation Stations can now be implemented uniformly by an associative search of the Updating Buffer entries only.

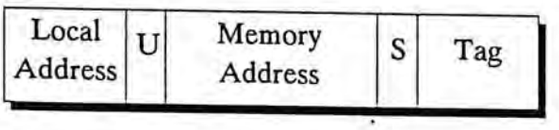


Figure 5.6. An Updating Buffer Entry

Every memory location with an outstanding load operation or store operation or both (either executing or pending for execution) will reserve an entry in the Updating Buffer. With reference to figure 5.6, the Tag field in each entry identifies the particular load or store operation (if it is a store, the Store Bit S will be set to 1, otherwise it will be 0) currently holding or going to supply the most updated value of the memory location referred by the Memory Address field. Let's refer to the illustration in figures 5.7, 5.9 and 5.10 where the following instructions are considered :

```
LD    F0,A[1]
MUL   F0,@10
ST    A[1],F0
```

When the Load instruction is decoded, the following steps are taken :

(1) By carrying out an associative search[3] over the Memory Address fields of all used Updating Buffer entries (those with the U Bit set to 1), we check if an entry for A[1]

[3] Alternatively, accesses to the Updating Buffer can also be implemented using Hashing with the Memory Address as the key. To safeguard performance, collisions should be properly handled. Limited overflow capability may be needed or we could hash to multiple Updating Buffer entries for a single address. A reasonable design may allow two addresses which are hashed to the same entry to coexist in the Updating Buffer. In this way, their corresponding memory locations can be accessed concurrently.

exists already.

(2) Since no entry exists, we therefore allocate one. Assume its Local Address is 38. At the same time, a Load Reservation Station is reserved.

(3) The Local Address 22 of the Load Reservation Station is written in the Tag field of the Updating Buffer entry allocated and the S bit is reset to signify that it is a load (see figure 5.7). The desired interpretation is that until further write to A[1] is encountered, its most updated content will be available upon the completion of the load operation manifested in the reservation station tagged 22.

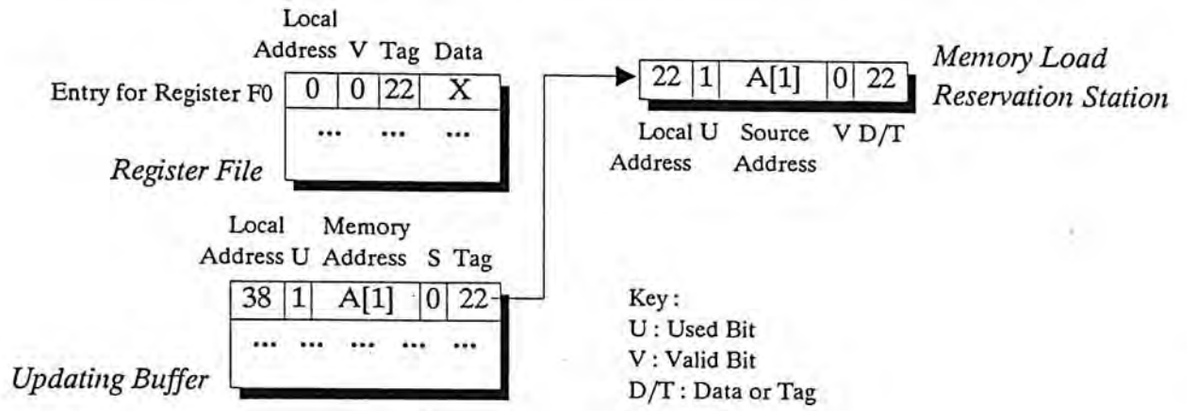


Figure 5.7. The Load instruction (tagged 22) will supply the most updated value of A[1]

As an illustration, suppose a second load from A[1] appears now, say to the register F1 :

(1) An entry for A[1] can be located in the Updating Buffer. As a result, no new load request will be submitted (in other words, we won't allocate a new Load Reservation Station for it).

(2) Stored in the Tag field of the Updating Buffer entry for A[1] is the Local Address 22 of the Load Reservation Station dedicated for reading A[1]. A copy of this is forwarded to register F1 as its source tag.

(3) When the data stored in A[1] finally arrives and appears in the Common Data Bus together with the tag 22, each of F0 and F1, with its matching tag, will gate in a copy of it.

The corresponding procedure graph transformation is illustrated in figure 5.8. Effectively, one memory access is saved in return.

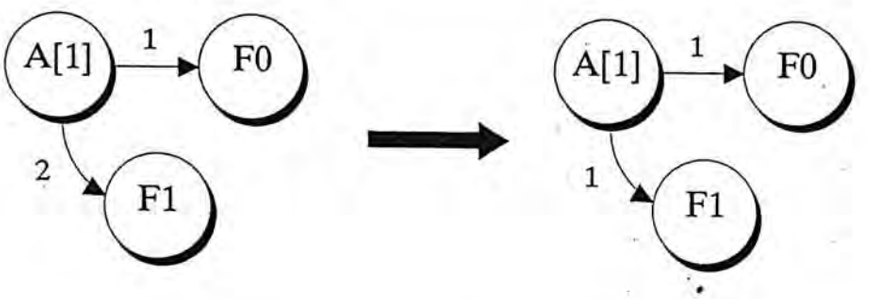


Figure 5.8. A Normalization of Pseudo Time Labels

Back to our example, the Multiply instruction is then decoded and the tag of the register F0 is modified to identify its new source (see figure 5.9). Finally, the Store instruction is encountered :

(1) A Store Reservation Station will be allocated, say the one with the Local Address 30. A Valid-Bit value of 0 in F0 indicates that the data to store is not ready yet. As a result, the tag 63 identifying the result of the multiplication F0←F0×10 is copied to the D/T field of the Store Reservation Station, and its Valid Bit is reset to 0 accordingly.

(2) Again, the Updating Buffer entry for A[1] is located (if none exists, allocate one). Its Tag field will become 30 and the S bit will be set to 1 (see figure 5.10). The rationale is that it is this store operation that will lead to the final content of A[1] and therefore, the original content of A[1] as loaded by the prior LD instruction (tagged 8) should no longer be visible to succeeding instructions.

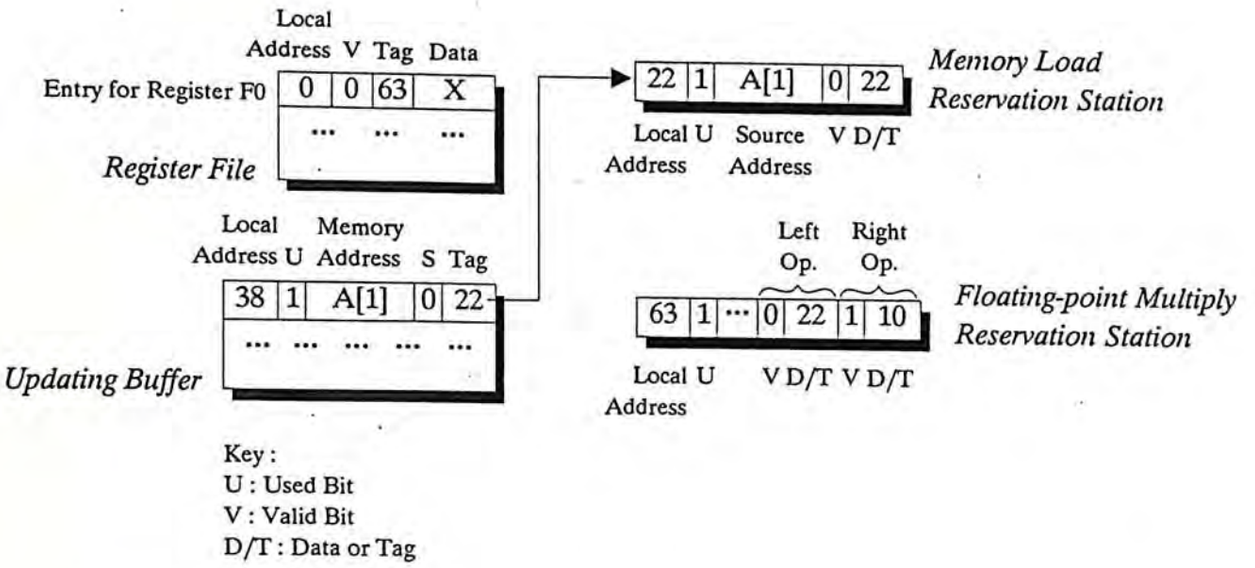


Figure 5.9. After the MUL is decoded, F0 waits for the result of F0←F0×10 which is tagged 63

Imagine the case when a load F2←A[1] is requested now. Having located the Updating Buffer entry for A[1], its S bit and Tag field together pinpoints the particular Store Reservation Station that will lead to the most updated value of A[1]. Suppose the data to store is still not ready then and the value of the Valid Bit is 0. As a result, the content of the D/T field is forwarded to F2.

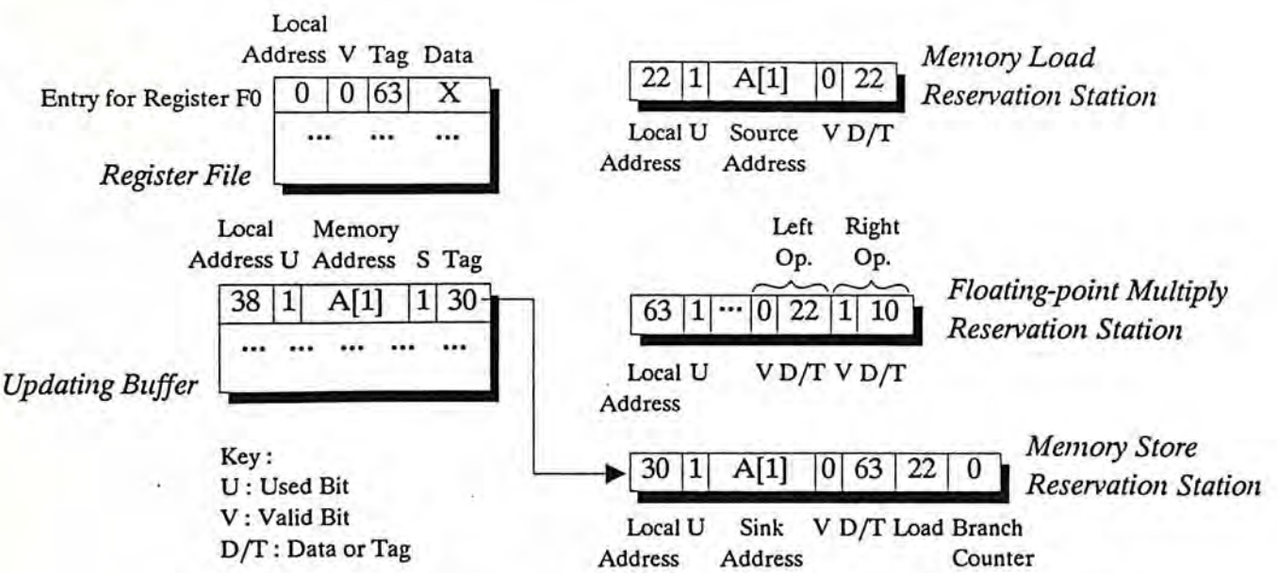


Figure 5.10. The Store operation (tagged 30) will define the final content of A[1]

When the multiplication finally completes, the tag 63 will accompany the result

in the CDB and a separate copy of it is shunted into the Store Reservation Station 30, the register F0 and the register F2. With respect to the underlying procedure graph, a Serial-to-Parallel Transformation has been achieved, as depicted in figure 5.11. Again, one memory access is saved in the process.

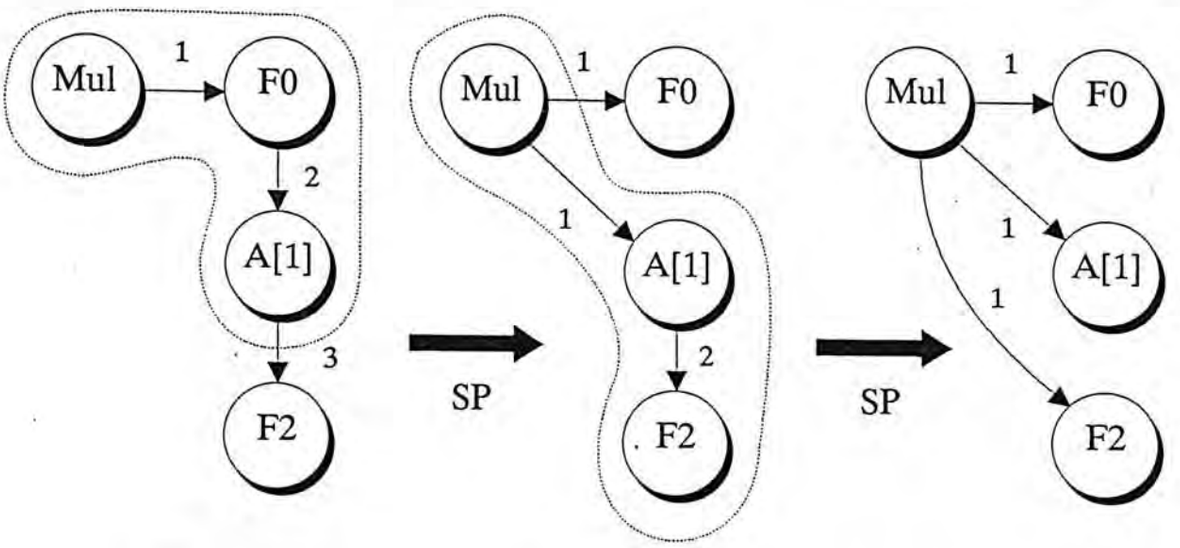


Figure 5.11. Two successive Serial-to-Parallel Transformations (SP)

The first SP is achieved when the store instruction A[1]←F0 is decoded while the second one is realized when the load instruction F2←A[2] is encountered. In the figure, the node labelled 'Mul' represents the floating-point Reservation Station with the Local Address 63 and the node labelled 'A[1]' denotes the Store Reservation Station 30.

If it happens that when the load operation F2←A[1] is decoded, the data to store in A[1] has been valid already, but yet, the store operation is either in progress or is pending for execution only, then F2 will receive a copy of the data itself instead of a tag[4]. In other words, a buffering effect is achieved and the convenience provided by this 'short-cut' in datapath will be available until the completion of the store operation.

As a conclusion, we can see that with the Updating Buffer, the concepts of

[4] If the D/T field of each entry of the Reorder Buffer is multiplexed for both data and tag, then the load access in this case will simply read out both the D/T and V fields regardless of the actual state of Valid Bit V. And for interpretation, if V=1, D/T manifests a data item. Otherwise, it is a tag.

tagging, forwarding and procedure graph optimization can now be applied effectively and extensively to memory data also. A 'homogeneous' architecture is achieved with memory accesses treated in much the same way as other functions.

5.4.2 Ordering and Consistency

In a superscalar design, an instruction is issued and fired as soon as all of its data dependencies and resource conflicts are resolved. While out-of-order issue, execution and completion are allowed, the actual order that instructions commit their effects in the storage locations should preserve the causality relationship spelled out in the program under execution (by the sequencing of the instructions). Any design should guarantee that the program is executed correctly, by which we mean that the final contents of all storage locations will remain the same under any uncertainty conditions (e.g. the actual rate of cache hit, etc). This is true for both memory and register data. Memory with its enormous storage space deserves a special policy.

5.4.2.1 Store After Store[5]

Two store operations S1 and S2 addressing the same storage location A should affect the memory in the correct order prescribed in the program, or inconsistency may result. With the Store Reservation Stations maintained as a strict first-in-first-out queue, their relative positions R1 and R2 will agree with the order of the corresponding instructions, i.e., R1<R2. As a result, S2 cannot proceed until S1 has finished even its data to store has become ready. Furthermore, the entry of A in the Updating Buffer will refer to R2 so that any succeeding reference to A will read the most updated copy. Please be noted that SSC is not applied for memory data.

---

[5] In the other two cases, namely Load-After-Load and Load-After-Store, the correct causality has been preserved by the Updating Buffer. For a Load-After-Load, the two requests will be simultaneously satisfied by a single memory read. And for a Load-After-Store, we simply 'short-circuits' the memory via the Store Reservation Station.

5.4.2.2 Store After Load

A store operation S to a memory location A should not be initiated until the prior load L to the same A, if any, has completed. In the Store Reservation Station R for S, the tag of the load reservation station dedicated for L, say T, will be stored, effectively holding its initiation. When L finishes, the data together with the tag T will be forwarded via the Common Data Bus. R with a matching tag T will have its blocking condition cleared. When it finally moves to the front of the queue of the reorder buffer and that the data to store has become ready, the store operation S can then proceed.

5.5 Speculative Execution

The performance potential of a superscalar processor such as the T-Architecture comes from the ability to execute multiple instructions simultaneously and in an out-of-order manner. To exploit the maximum efficiency of its duplicated computing resources, there must be enough instruction-level parallelism to feed the wide bandwidth of the superscalar pipeline. While this issue is mostly application-dependent (see [Jouppi89] and [Jouppi&Wall88]), an efficient design should be capable of extracting the maximum amount of parallelism available.

5.5.1 Procedural Dependencies

The larger the examined scope of the algorithm, the more effective the optimization achieved (equivalently, a larger procedure graph should in general allow more transformations to be applied, leading to more fruitful results). Thus we duplicate the instruction fetch unit and the decoder. Besides, an instruction window is implemented by adopting a number of reservation stations. The sole objective is to allow multiple instructions to be considered at the same time. But that is not the whole story.

The presence of procedural dependencies as created by conditional branch

instructions lowers performance in the following ways :

- The normal sequential instruction fetch sequence will be corrupted. Consequently, the fetch efficiency (in terms of the number of instructions fetched per cycle) will be partly or completely sacrificed.

- Sometimes, the outcome of a branch instruction may depend on the result of a preceding instruction. During this branch delay period, the address of the target instruction cannot be known and instruction fetch ceases. The degree of overlapping of execution will be decreased as a result.

- Other factors such as target instruction alignment further lower the fetch efficiency. Interested readers should refer to [Smith et al.89] for a detailed discussion.

Several techniques have been found effective for tackling procedural dependencies. The simplest approach is to freeze the fetch unit every time a conditional branch instruction is encountered until the corresponding procedural dependency is resolved and the correct target located. But then we will lose much performance. Alternatively, we can schedule (arithmetic) instructions back and forth around a branch instruction, to shorten its branch delay (software optimization techniques such as the delayed branching belong to this category, see [Hennessy&Patterson90] for details). Finally, we can also predict the outcome of a branch and allow the execution to continue in the predicted path in a conditional mode (see [Smith et al.90]). Each of the methods has its pros and cons, and the T-Architecture has adopted a combined strategy.

5.5.2 Branch Instruction Format

In the T-Architecture, we have chosen not to design condition code. Instead, a conditional branches on the value (negative, zero or positive) of a specific (floating-

point or integer) register. As a result, branch instructions assume the following formats :

	Format	*Interpretation*	
BZ	**Ri, Address_if_taken**	**PC←Address_if_taken**	**if (Ri) = 0**
BG	**Ri, Address_if_taken**	**PC←Address_if_taken**	**if (Ri) > 0**
BL	**Ri, Address_if_taken**	**PC←Address_if_taken**	**if (Ri) < 0**

5.5.3 Branch Prediction

The simplest branch prediction mechanism is used. With reference to figure 5.1, each active branch instruction is allocated an entry in the Branch Predicting Buffer. A branch, if is first encountered, will be assumed to be taken (thus fetch and execution will continue from the instruction at Address_if_taken). Otherwise, the destination address it has last taken (which is stored in its Branch Predicting Buffer entry) will be used as the predicted new PC.

5.5.4 Branch Instruction Unit

As shown in figure 5.1, there are N entries in the Branch Instruction Unit. Each entry can hold an active branch instruction. Therefore, a maximum of N branch instructions can be outstanding at any time. Alternatively, we may say that the maximum level of branching is N.

From another point of view, these entries serve as the reservation stations for the branch functional unit effectively. A branch instruction buffered is allowed to be fired and executed as early as possible in an out-of-order manner when all of its data dependency has been resolved. Yet, they must be committed in the same order as they are encountered/decoded (this minimizes the number of combinations). In other words, the entries of the Branch Instruction Unit will be managed as a strict first-in-first-out queue. In each cycle, the branch instruction at the front (if any) will be examined. If it

has been completed, the actual outcome will be compared against the predicted destination. A match reveals a successful prediction and a BRANCH_PREDICTED signal will be generated and broadcasted system-wide. Otherwise, a BRANCH_REPAIR signal is produced instead and the corresponding entry in Branch Predicting Buffer will have its Destination Address field replaced by the correct outcome.

5.5.5 Register Backups

A consequence of allowing speculative execution via branch prediction is that some instructions can be wrongly executed which modify the machine state incorrectly. Should a branch turn out to be mistakenly predicted and executed, we have to recover the correct machine state just before it and resume execution from the instruction at the correct target. A crucial part of this 'machine state' will be the contents of various storage locations - registers and memory. In the T-Architecture, a hybrid scheme has been implemented. Register and memory data are handled differently. In this section, we consider the former first.

The checkpoint repair [Hwu&Patt87] method has been adopted for handling register data. As shown in figure 5.12, N backups have been adopted for the floating-point and integer register file respectively (N being the maximum level of branching). By allowing as many as N branch instructions to be outstanding at the same time, each backup serves as a checkpoint (or logical space) with respect to the corresponding conditional branch. With out-of-order execution and lookahead, the Register File always maintains the lookahead state of the machine.

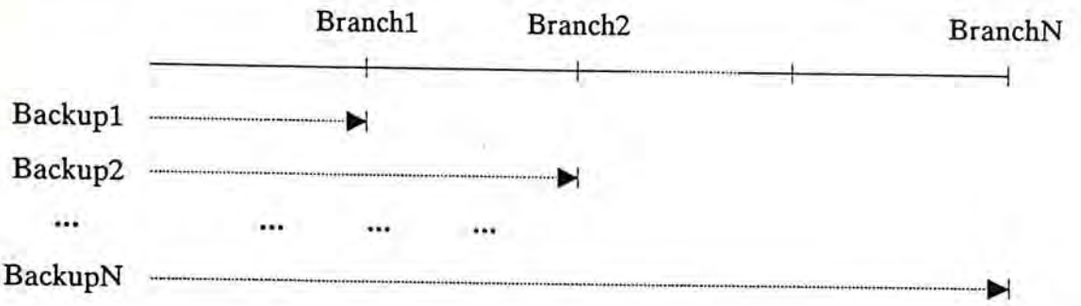


Figure 5.12. Each Backup serves as a checkpoint w.r.t. the corresponding branch instruction

With reference to figure 5.13, when a branch instruction is decoded, the current branch level is incremented by 1. A free backup (with a value of 0 in its Branch Level Counter) will be allocated with its Branch Level Counter updated by the current branch level. If it is at the first branch level (current branch level=1), the local Address of the backup will be stored in the First Level Backup Counter. In any case, a "snapshot" of the registers' values will be copied to the backup as a part of the correct machine state before the branch instruction. Only instructions encountered before this conditional branch can affect this backup. The machine then continues its conditional mode of execution from the instruction at the predicted target.

5.5.5.1 Branch is Correctly Predicted

In each cycle, the first level branch instruction (if any) which has completed execution will be committed. Suppose its actual outcome agrees with our prediction. Upon receipt of the BRANCH_PREDICTED signal, the backup corresponding to this first level branch instruction (with Local Address referred by the First Level Backup Counter) will be discarded by simply resetting the value of its Branch Level Counter to 0. At the same time, the Branch Level Counters of other backups (if the current branch level is greater than 1, i.e. more than one branch instruction are outstanding) are decremented by 1, and so is the current branch level. As a result, the backup (and the respective branch instruction) originally at the second level will now move to the first level, with its Local Address written in the First Level Backup Counter.

5.5.5.2 Branch Repair

If it happens that the first level branch instruction is mistakenly predicted and executed, an exceptional condition will be raised by broadcasting a BRANCH_REPAIR signal. The following actions are taken in response :

- The Instruction Fetch Unit and the Decoder are flushed and stalled.
- To restore the correct machine state before the faulty branch[6], its corresponding backup B as addressed by the First Level Backup Counter will be used. The content of each register in the Register File is corrected by the respective entry in B.
- All backups (including B), if allocated, are then discarded and the current branch level is reset to 0.
- All entries in the Branch Instruction Unit will be flushed.
- Instruction fetch restarts from the correct target. Normal execution resumes.

5.5.5.3 Example

Consider the following code sequence :

```
            LD    F0,A[1]
            BL    F0,TARGET2
TARGET1:    MUL   F0,@3
            ...   ...
TARGET2:    ADD   F0,@3
            BG    F0,TARGET3
            ...   ...

TARGET3:    MUL   F0,@9
            ADD   F1,@1
```

[6] From another point of view, the register repair mechanism recovers the correct causality relationship, effectively undoing the effects of those wrongly executed Procedure Graph Transformations and instructions.

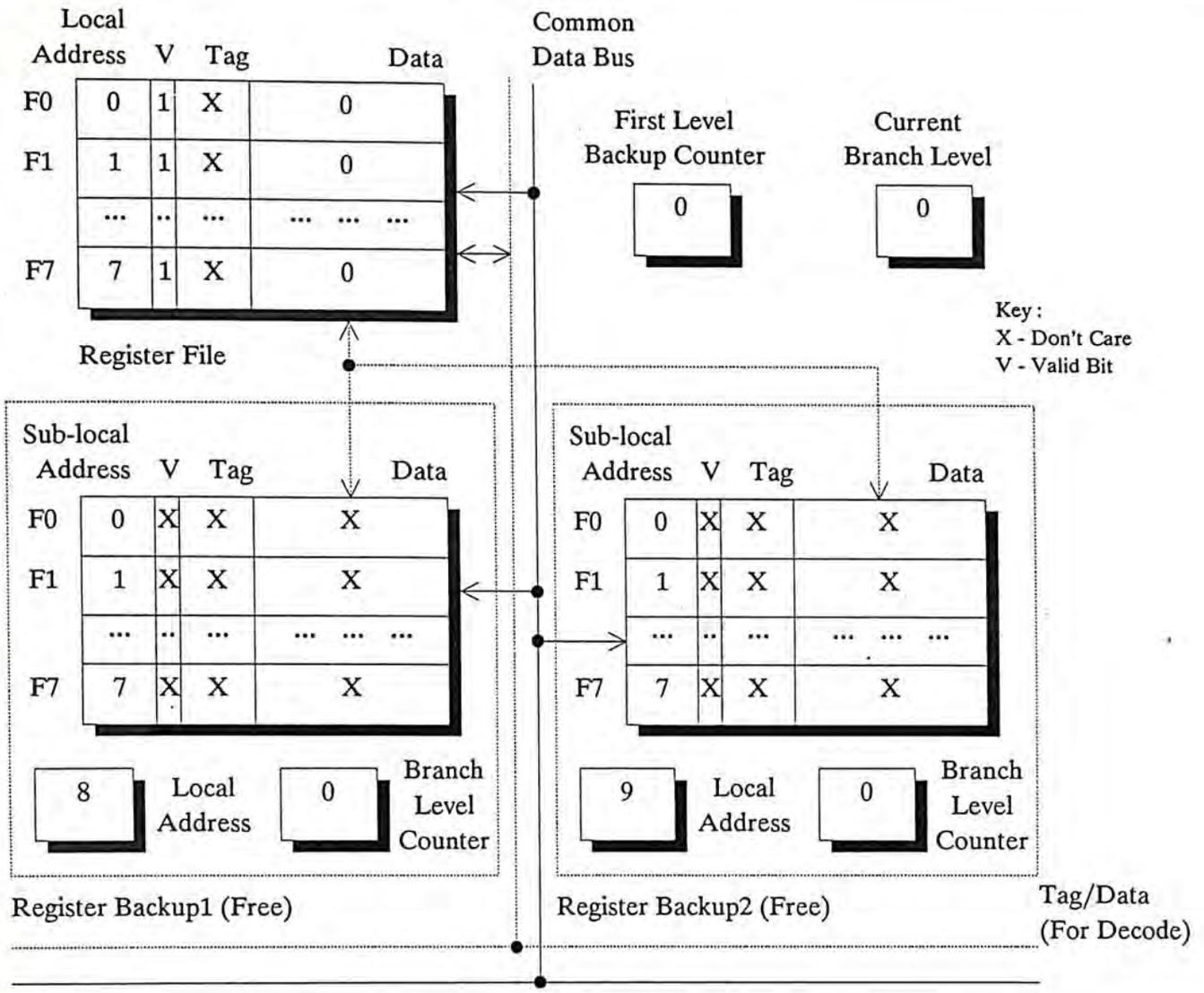


Figure 5.13. The structure of the Register Backups (for floating-point registers only)

When the LD instruction is decoded, a Load Reservation Station will be allocated. Its Local Address 22 will be written in the Tag field of the F0's entry in the Register File, which effectively identifies the Load Reservation Station with Local Address 22 as its source.

Assume that the branch instruction (BL) is at the first branch level (as depicted by the Current Branch Level). A backup (Backup1) will be prepared for the whole (Floating-point and Integer) Register File. Please be reminded that all of the Data field, the Valid Bit and the Tag field will be saved. As shown in figure 5.14 the Tag field of the F0's entry in the register backup is set to 22 also.

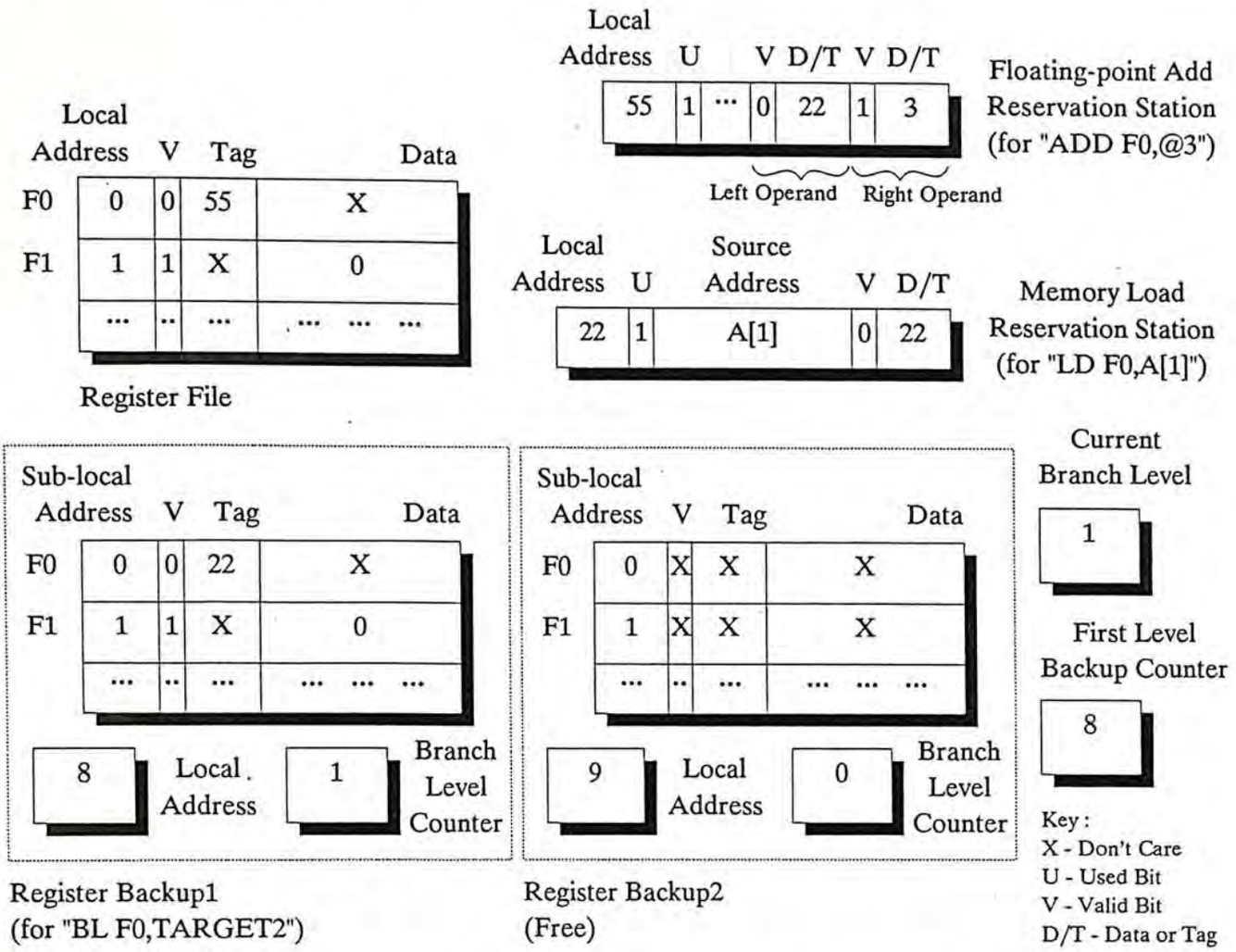


Figure 5.14. Backup1 is used as a checkpoint for the branch instruction "BL F0,TARGET2"

(Current Branch Level = 1 and First Level Backup Counter Records the Local Address 8 of Backup1)

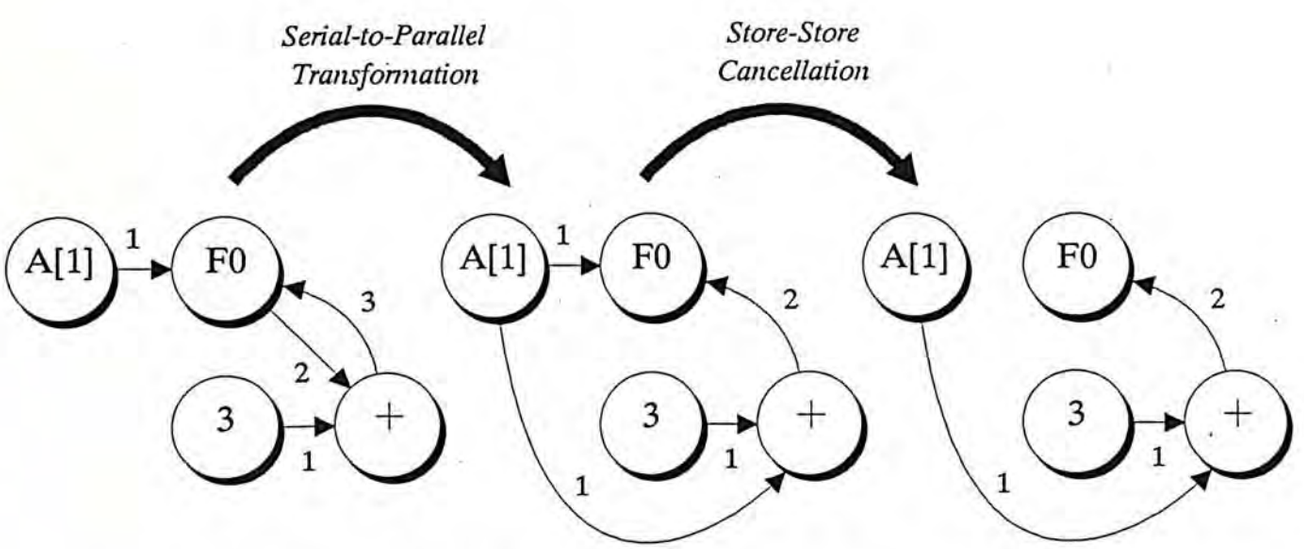


Figure 5.15. Conditional procedure graph transformations across basic block boundary

(When the instruction "ADD F0,@3" is decoded)

Branch prediction results in conditional execution being continued from the ADD instruction labelled by TARGET2. An ADD Reservation Station is dedicated for this floating-point addition. Its Local Address 55 overwrites the original value of the Tag field of the F0's entry in the Register File (but the corresponding entry in Backup1 will remain unchanged). A Serial-to-Parallel Transformation and a Store-Store Cancellation are applied in turn (as illustrated in figure 5.15).

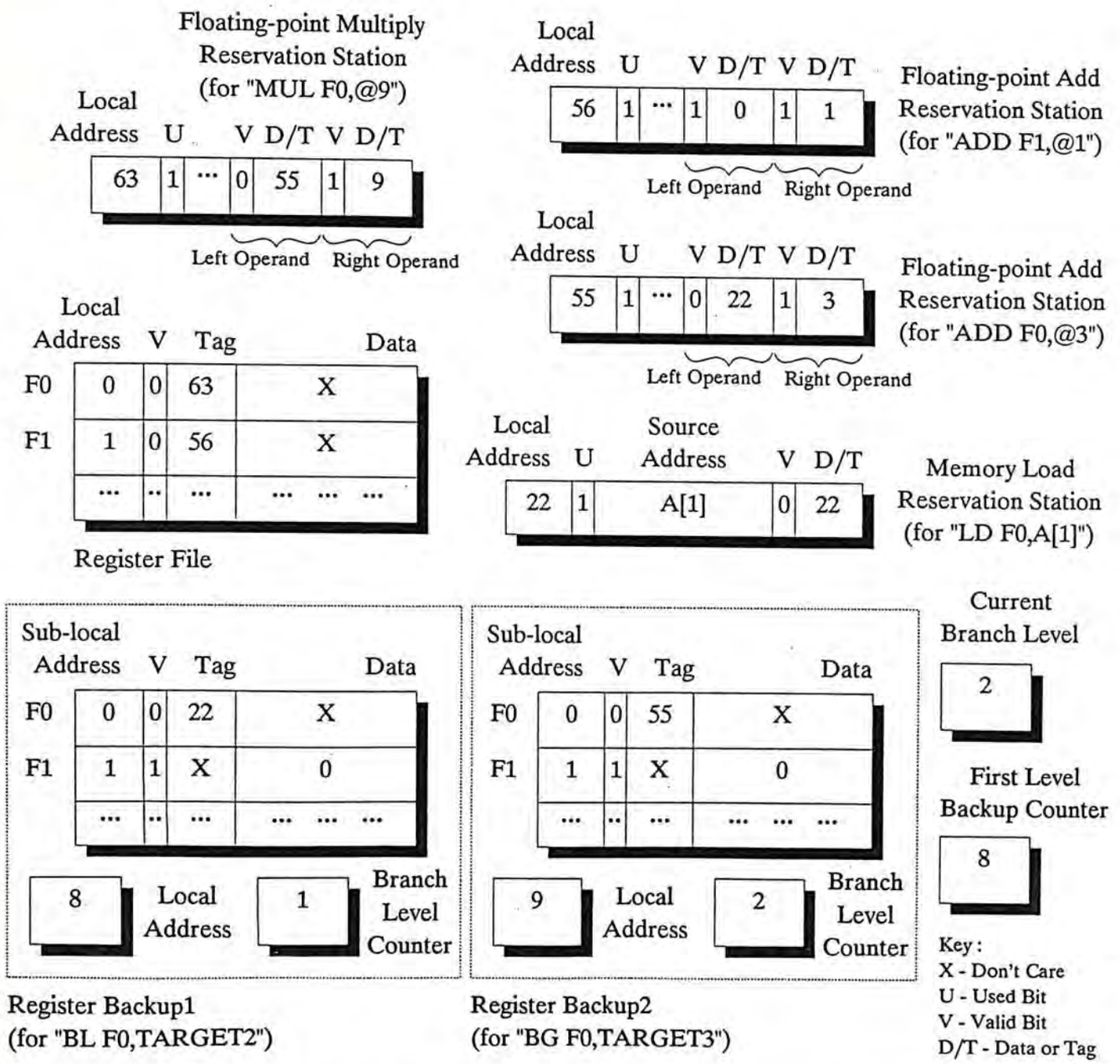


Figure 5.16. Backup2 is used as a checkpoint for the branch instruction "BG F0,TARGET3"

Then comes the BG instruction. Again, a backup (Backup2) will be made to

prepare for the possible branch fault. Its Branch Level Counter is set to 2 since the BG instruction is at the second branch level. Conditional execution then continues from the MUL instruction at TARGET3 and the MUL Reservation Station with Local Address 63 is allocated. By updating the Tag field's content of F0's entry in the Register File with this Local Address 63, another Store-Store Cancellation is achieved (see figures 5.16 and 5.17) and the final content of F0 will be originated by this multiplication. Thus, we can see that the Register File always maintain the lookahead state of the machine.

Suppose the execution of the BL instruction completes first and its evaluation reveals that our prior prediction for it was correct. Upon receipt of the BRANCH_PREDICTED signal, Backup1 (with its Local Address 8 recorded in the First Level Backup Counter) will be discarded. Backup2 and its corresponding branch instruction (BG) now moves to the first level (see figure 5.18).

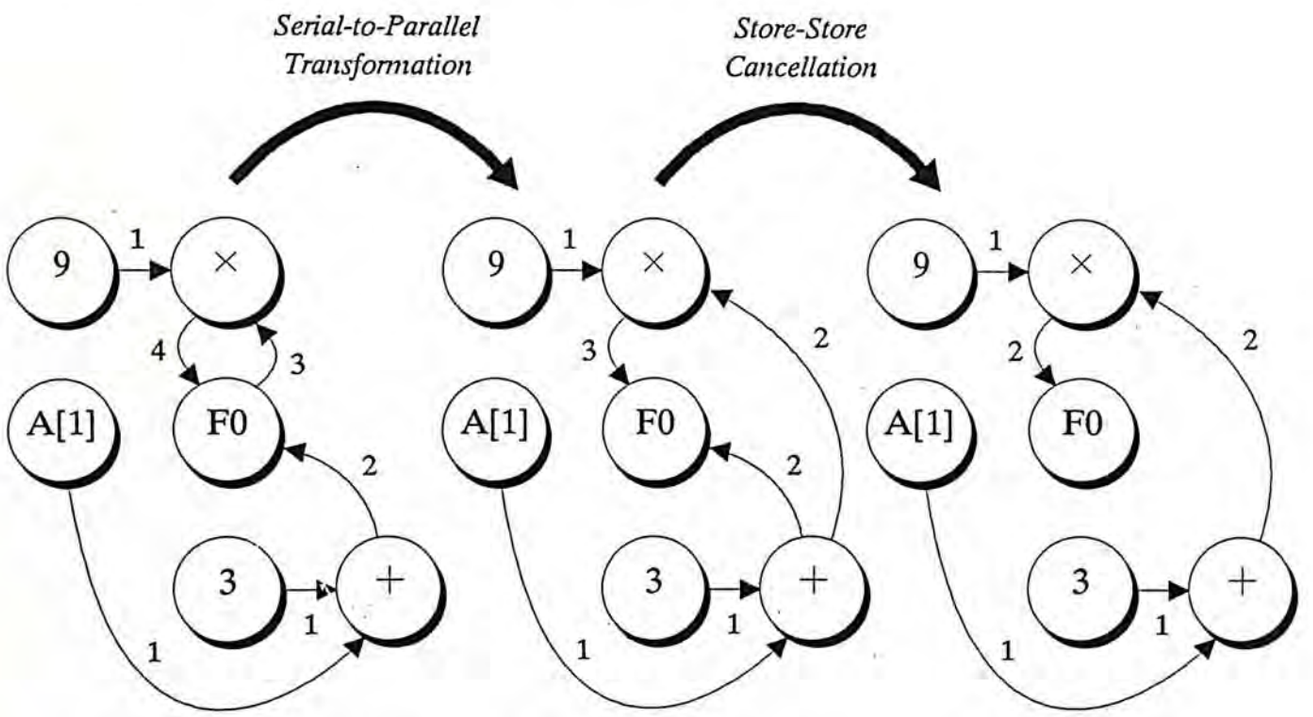


Figure 5.17. Conditional procedure graph transformations across basic block boundary

(When the instruction "MUL F0,@9" is decoded)

The completion of the instruction "ADD F0,@3" (labelled by TARGET2)

initiates the execution of the BG instruction. Suppose we have a successful prediction again. A a result, Backup2 will be discarded. The Current Branch Level and the First Level Backup Counter are both reset to 0. All procedural dependencies have been resolved.

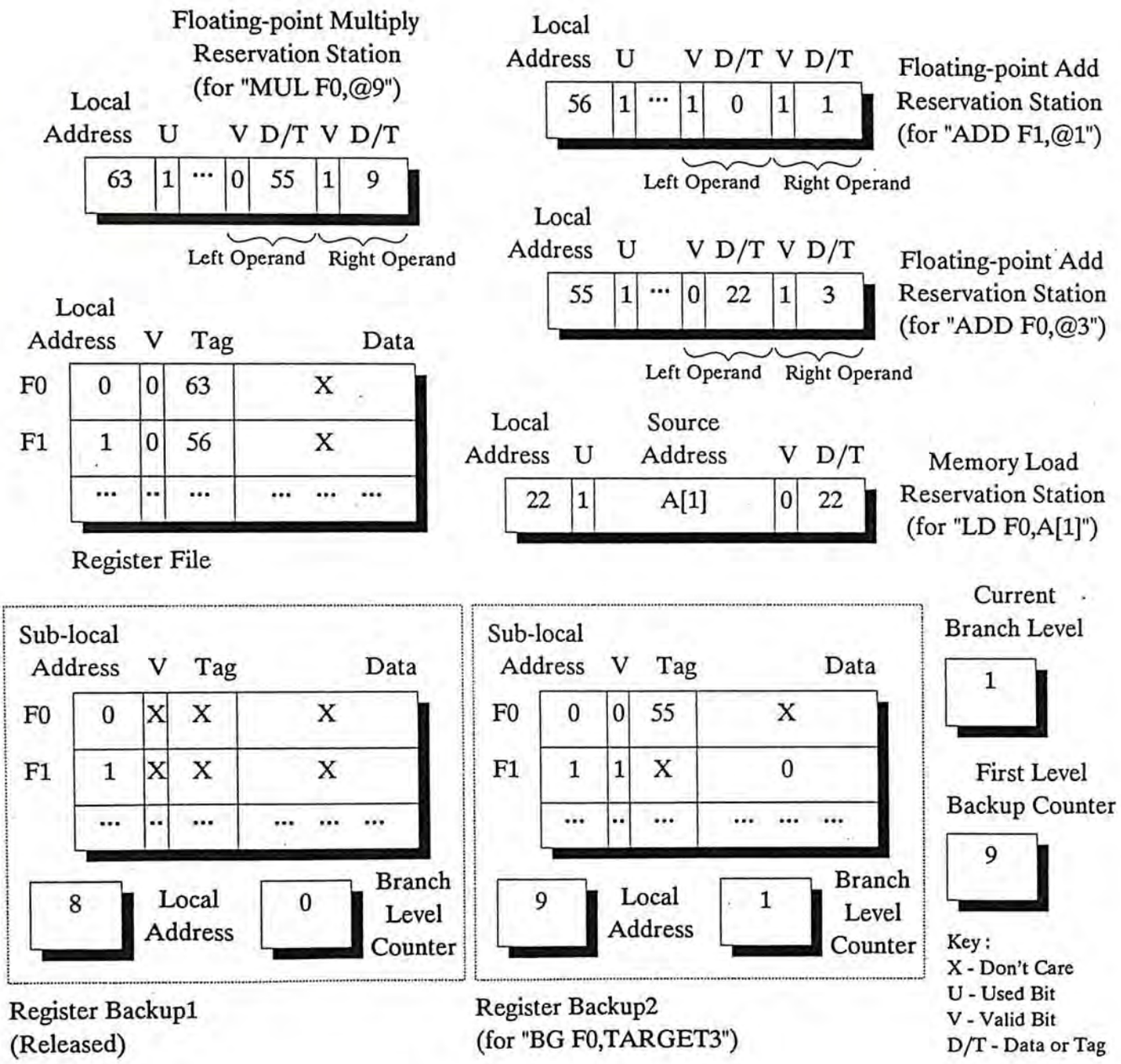


Figure 5.18. Branch Prediction for "BL F0,TARGET2" was correct. Backup1 is discarded

Backup2 and "BG F0,TARGET3" now moves to the First Branch Level, as indicated by the First Level Backup Counter and the Current Branch Level.

Two points should worth further consideration. First, with Speculative Execution,

we are no longer constrained to do strict local optimizations only. Procedure graph transformations can now be applied across basic block boundary. For a maximum branch level of N, up to N+1 basic blocks can be optimized together (in our example, the Store-Store Cancellation in figure 5.17 is in fact applied across two basic block boundaries). The increase in the average run length[7] between branches (that stall fetch and decoding) makes available a wider scope of candidate instructions (or a larger basic block) to optimize. Global optimization as allowed by the maximum level of branching should lead to more fruitful results.

However, the effects of these transformations are temporary only and they cannot be committed until the corresponding procedural dependencies are resolved. By keeping backups for the machine states, we can undo incorrect transformations in case of a branch fault and restore the legitimate precedence relationship.

In addition, Speculative Execution effectively saves the decoder from being stalled frequently. On the one hand, the fetch efficiency will be promoted. More importantly, lookahead allows the out-of-order executions of succeeding ready-to-go instructions. As illustrated by our example, the instruction "ADD F1,@1" can be fired immediately, overriding the procedural dependencies as a result. If our branch prediction has a high hit rate, expeditions of this kind will realize significant speedup. In fact, this agrees with our belief that the degree of overlapped execution should only be limited by Read-After-Write (RAW) or true data dependency, and the architecture must try to such unnecessary delays as those incurred by procedural dependencies.

Suppose we have a branch fault for the BL instruction in fact. The machine state and the correct causality relationship are restored by copying corresponding entry of Backup1 to the Register File, effectively undoing the effects of those wrongly executed

[7] [Johnson91] defines run length as the average number of instructions dynamically executed between the decoding of two conditional branch instructions. Alternatively, it can be perceived as the average size of a basic block.

instructions and transformations. All procedural dependencies are resolved and both Backup1 and Backup2 will be released. Instruction fetch and decoding then resumes from TARGET1 (see figure 5.19).

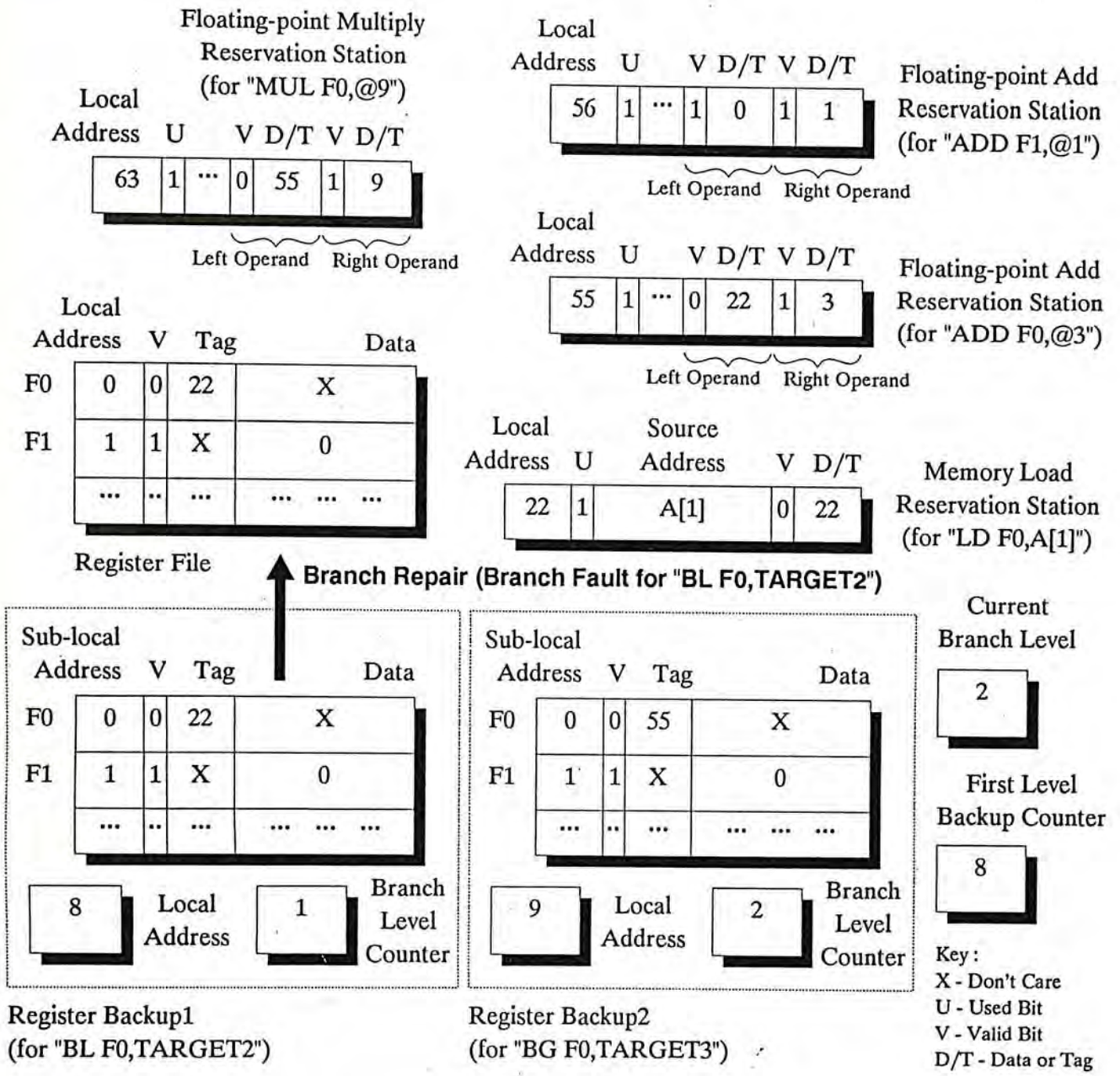


Figure 5.19. Branch Repair using Backup1

The First Level Backup Counter helps to locate Backup1 (With Local Address 8) for restoring the Register File. After the Branch Repair, Backup1 and Backup2 will be released. At the same time, the Current Branch Level and First Level Backup Counter will both reset to 0. No procedural dependency is outstanding now.

As a final comment, one can see that we don't have to flush out the issued

operations in the Multiply Reservation Station 63 and the Add Reservation Station 56 (as revealed by the branch fault, their respective instructions should not appear in the correct execution path). The reason is that although they are allowed to execute to completion, their effects will not be committed. By restoring the Tag fields from Backup1 also, the backward pointers 63 and 56 originally in F0 and F1 respectively will be overwritten. As a result, upon completion of these two instructions, there would be no recipient demanding for them (even if they are placed on the Common Data Bus) and the contents of F0 and F1 will remain unchanged, preserving the correct causality relationship.

Doubtless, the use of backward pointers for representing data transfer arcs has greatly simplified the branch repair mechanism. Precisely, we don't need to associate procedural dependency information with the reservation stations and/or the functional units, nor do we have to search and "evacuate" all incorrect instructions in case of a branch fault. The rationale is that it would be much easier to avoid committing their effects than to prevent them from execution.

5.5.6 Total Ordering Memory Stores

The use of a hardware backup as applied to register data is impractical for the memory because of its huge addressing space involved. Adopting a similar approach as found in the Reorder Buffer proposed in [Smith&Pleszkun88], our architecture dictates that all memory store operations to be held until the corresponding procedural dependencies have been removed.

The execution of a store instruction may suffer from considerable delay as a result. In spite of this, we claim that the overall performance will not be sacrificed much. On the one hand, the arithmetic instruction(s) which produce(s) the data to store should have been finished before the store is ready to be fired. In addition, a succeeding

instruction (such as a load) which refers to the sink of this store as its source operand can read out the data from the store reservation station concerned as soon as its content becomes valid, effectively 'short-circuits' the memory. As a result, the fact that a store operation (with its data to store has been valid already) becomes held in a store reservation station due to procedural dependencies will not block other operations[8].

In general, if the maximum branch level is N, a separate Branch Counter of $\lceil \log_2 N \rceil$ bits will be included in each Store Reservation Station. When a store operation S is decoded with the Current Branch Level equal to N', the Branch Counter in its corresponding Store Reservation Station will be initialized by the value N'. Every time when the evaluation of the outermost branch instruction (at the first branch level) reveals that the prior prediction for it was in fact correct, the Branch Counter of each pending store will be decremented by one.

As discussed earlier, the Store Reservation Stations are organized as a first-in-first-out queue. At any time, a store operation at the front of this queue with the data to store ready should also wait for its Branch Counter to become zero before it can be fired. Should a branch fault occur, all pending stores with a non-zero value in its Branch Counter will be flushed out.

Consider the example depicted in figure 5.20(a) where the maximum branch level is 2. When the first branch instruction Br1 is decoded, the Current Branch Level becomes 1, and the initial value of the Branch Counter corresponding to the store operation St1 will be 1 also. Without exception, Br2 is handled similarly and the value of the Branch Counter for St2 will be initialized to 2. St0 is decoded before any branch instruction is encountered, thus its Branch Counter is zero. The situation is depicted in figure 5.20(b). As shown, only St0 can proceed (provided that other conditions are

[8] The situation is different for memory loads. Succeeding instruction(s) usually need to operate on the memory data read by the preceding load instruction(s).Thus the delays of the latter will propagate to the former.

satisfied).

Suppose Br1 finally completes which reveals that our prior prediction for its target was correct, the Branch Counters of St1 and St2 will be decremented by one, effectively resolving the procedural dependency of St1. The resulting situation is shown in figure 5.20(c). If it turns out that we have a branch fault for Br2, all memory store operations encountered after the faulty branch will be flushed out. Thus St2, because of its non-zero Branch Counter, will be deleted.

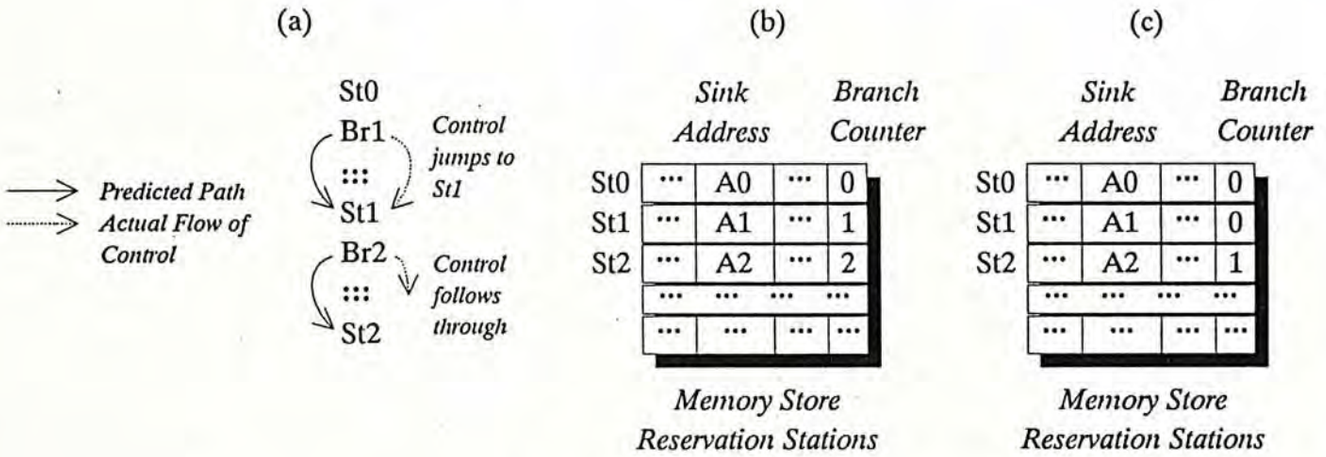


Figure 5.20. Representing Procedural Dependencies using Branch Counters

5.5.7 Simplifying the Checkpoint Repair Mechanism

Both the hardware and the control logic can be much simplified if we restrict ourselves to single branch level only. In other words, at most one branch instruction can be outstanding at any time and the second one will be held upon decoding.

First of all, only one backup is necessary now and it won't need its Branch Level Counter any more. This could be a significant saving in circuitry if the number of registers is large. Besides, we no longer have to record the Local Address of the First Level Backup because there is only one backup to use in case of a branch repair. The Current Branch Level Counter can now be replaced by a single bit (the Speculative Executing Bit) which when set, indicates that there is a branch instruction outstanding.

Until its procedural dependency is resolved, the second branch instruction will be held in the decode stage. When a branch is correctly predicted, we simply discard the backup by clearing the Speculative Executing Bit.

On the other hand, by restricting ourselves to single branch level only, the Branch Counter in each Store Reservation Station can be replaced by a single bit (the Branch Bit). In case a branch instruction is active when a store operation is decoded/issued, the corresponding Store Reservation Station will have its Branch Bit set to 1. Upon receipt of an BRANCH_PREDICTED signal telling that the branch was correctly predicted, all Branch Bits which are currently set will be cleared, thus resolving the procedural dependencies involved.

There is no denying that the amount of parallelism to exploit will be decreased at the same time because optimization can now be applied to a narrower scope only. However, with a short latency of the branch instruction (say two cycles), performance loss due to the blocking of a branch instruction can be minimized. Doubtless, the use of various instruction scheduling techniques such as delayed branching can further shorten the average branch delay.

5.6 A Simulator for the T-Architecture

A simulation program has been devised using the C programming language in the PC environment. Performance of the T-Architecture is being measured in terms of the average number of instructions completed in each cycle (IPC). The simulator is carefully parameterized to enhance the study of the effects of different factors on performance. These factors would probably include the fetch size, the bandwidth of the common data bus and the performance of the cache. They interact in such a way that a continuum of design combinations should be present. We hope that the simulation results given later in section should provide insights leading to wise design decisions.

5.6.1 Basic Configuration of the Simulator

- Number of Functional Units and Reservation Stations

	Add (Floating-point)	Mul (Floating-point)	Add (Integer)	Mul (Integer)	Branch	Shift	Logic	Memory Load	Memory Store
Number of Functional Units	1	1	1	1	1	1	1	1	1
Number of Reservation Stations	8	8	3	3	1	1	1	8	8

Note :

- All the functional units are pipelined.
- The number of Updating Buffer entries is 8.

- Latencies of Operations :

Add (Floating-point)	Mul (Floating-point)	Add (Integer)	Mul (Integer)	Branch	Shift	Logic	Memory Load	Memory Store
3 Cycles	4 Cycles	1 Cycle	3 Cycles	2 Cycles	1 Cycle	1 Cycle	2 Cycles	2 Cycles

Note :

- As speculative execution is adopted in our design, should a branch instruction be mistakenly predicted and executed, the architecture should be capable to roll-back to the correct machine state just before the faulty branch, and then to continue its execution from the correct path. This addresses the need to restore the contents of the whole set of floating-point registers and integer registers from the corresponding backups. One extra cycle is dedicated for this branch repair activity.

- The latency of a memory load or store operation applies only when the storage location of interest can be located in the cache. Should a cache miss occur, one will suffer from a penalty of 20 cycles which accounts for the access to the main memory and the updating of the cache (for any block replacement and memory

write-back, if necessary).

- Maximum Branch Level for Speculative Execution
  - A maximum branch level of N means that up to N conditional instructions can be outstanding at any time, each residing in a separate entry of the Branch Instruction Unit (which are either executing, or have been evaluated but their effects are not committed yet, or are waiting for their data dependencies to be resolved before they can be fired). The (N+1)-th branch instruction encountered will be held upon decoding until the branch instruction at the first level has completed execution and have its effect committed.

    For the basic configuration, we restrict the simulator to have a maximum branch level of one only. In this way, hardware can be simplified much, e.g. only a single set of register backup is necessary and that a single bit will suffice for the Branch Counter field of each Store Reservation Station.

5.6.2 Parameters of the Simulator

Three dimensions of design alternatives are considered and their descriptions follow.

• **Fetch Size (F)**	It addresses the maximum number of instructions which can be fetched and decoded in each cycle. The fetch size dictates the maximum performance of any system. Take for instance, if only two instructions are fetched and decoded in each cycle (i.e. F=2), the maximum number of instructions processed per cycle cannot exceed two (i.e. IPC≤2).

• **Bandwidth of CDB (B)**	This refers to the number of results (functional units' outputs) that can be distributed system-wide to multiple destinations in each cycle.
• **Cache Hit Ratio (C)**	It denotes the probability that a memory location involved in a load or store operation can be located in the cache. As we are only interested in the influence of the performance of the cache on the efficiency of the whole system, we choose to avoid the tedium involved in simulating the actual operations of the cache. Instead, we will model its performance using a random number generator[9]. Every time when a memory store or load operation is to be initiated, a random number r will be obtained where $0 \leq r \leq 1$. Suppose that a cache hit ratio of 90% is expected. Then if $r > 0.9$, a cache miss occurs.

They manifest themselves as parameters of the simulator. The impact on the total performance is assessed by varying each in turn while fixing the others.

5.6.3 Benchmark Programs

Experiments have been carried out on the simulator using three benchmark programs. They are : (a) the Gaussian elimination inner loop involving 30,000 floating-point elements, (b) the Matrix Multiplication $C=A \times B$ where A, B and C are all 100×100

[9] The random number generator adopted in the simulator is an implementation of the algorithm proposed in [Wichmann&Hill87].

square floating-point matrices, and (c) the Prefix Sum of an array of 30,000 floating-point numbers. Their corresponding program segments have been shown in figure 5.21 for reference.

```
MOV   R0,@100          MOV   R0,@30000
MOV   R1,R0            MOV   R1,R0
IADD  R0,@-1           LD    F0,A[R0]
MOV   R2,@100          IADD  R0,@-1
MOV   R3,R2            MUL   F0,F1
IADD  R2,@-1           ADD   F0,F2
MOV   F0,@0            ST    A[R1],F0
MOV   R4,@100          BG    R0,@1
MOV   R5,R4
IADD  R4,@-1           b. Gaussian Elimination
LD    F1,A[R1][R5]
LD    F2,B[R5][R3]
MUL   F1,F2            MOV   R0,@3
ADD   F0,F1            MOV   F0,@0
BG    R4,@8            MOV   R1,R0
ST    C[R1][R3],F0     LD    F1,A[R0]
BG    R2,@4            IADD  R0,@-1
BG    R0,@1            ADD   F0,F1
                       ST    A[R1],F0
a. Matrix Multiplication BG  R0,@2

                       c. Prefix Sum
```

Figure 5.21. The three benchmark programs

These benchmark programs are carefully chosen in such a way that the varying amount of instruction-level parallelisms in them could to certain extent, help to minimize the effects of those application-dependent factors on performance (For a detailed discussion of the interference on performance measurement due to the nonuniform distribution of parallelism in different applications, please refer to [Jouppi89] and [Jouppi&Wall88]).

The preliminary results shown in figure 5.22 should justify our choice. They are obtained by assuming the basic configuration for the simulator with both the fetch size and the bandwidth of the CDB equal to one. The cache hit ratio is kept at 90%.

A remarkable characteristics possessed by all of these benchmarks is that they are quite 'raw' - by which we mean little or none software optimization has been carried out on their codes. Neither loop unrolling nor delayed branching has been applied to deal with the procedural dependencies implied by those branch instructions. If only minor software optimizations are incorporated, the measured performance would not

be unrealistically exaggerated, so that we can focus on the speedup brought about by the various features of our design - procedure graph optimization, superscalar techniques, tagging and forwarding, etc. We do not mean to rule out the use of software optimization techniques, but we are confident that the unique hardware characteristics of our design alone suffice to realize significant performance improvement.

As a final note, since the operation of the cache is only modeled by using the random number generator, we can only assume an expected cache hit ratio for the simulator. Yet, the actual outcome may vary, and as we will see, the performance of the architecture is quite sensitive to the actual cache hit ratio. To make things more precise, we will complete five runs for each experiment, with their average taken for the final result. Just for reference, the average cache hit ratio will also be included each time.

	Matrix Multiplication	Gaussian Elimination	Prefix Sum
Dynamic Instruction Count	7,090,283	210,003	180,004
Instructions Per Cycle	0.841544	0.888058	0.826560
Cache Hit Ratio	90.005%	90.065%	90.091%
Memory Loads + Stores	2,000,000 + 10,000	30,001 + 30,000	30,001 + 30,000

Figure 5.22. Amount of parallelism and number of memory accesses in the three benchmarks

5.7 Experiments

Experiments have been devised to study the following issues :

- The effects of the Cache Hit Ratio on performance
- The benefits of using Reservation Stations
- The effects of the Fetch Size on performance
- The bottleneck of the Common Data Bus

5.7.1 Experiment 1

- *Objective* To study the effects of varying the Cache Hit Ratio on the overall performance of the T-Architecture.

- *Conditions* Assuming the basic configuration for the simulator and that the bandwidth of the CDB is kept at two results per cycle, three cache hit ratios - 85%, 90% and 95%, are evaluated in turn. In each case, the performance is measured for three different fetch sizes - one, two and four instructions per cycle.

- *Results* We summarize the results in figure 5.23. As expected, the performance of the cache dictates the overall efficiency of the architecture. If only one instruction is fetched and decoded in each cycle, a 15.165% of performance promotion can be realized by increasing the cache hit ratio from 85% to 90%. Another 9.297% of incremental improvement can be achieved by further promoting the cache hit ratio to 95%, attaining the performance level of about 0.966 instructions per cycle as a result. In other words, the architecture is operating at over 96% of the maximum efficiency[10].

The influence of the cache hit ratio on performance becomes more profound when the degree of parallel or overlapped execution is increased. When the fetch size is two instructions per cycle, a 26.423% of performance gain is obtained by increasing the cache hit ratio from 85% to 90%, which if further promoted to 95% will bring along an extra 28.295% of performance improvement. More importantly, we can see that its effect has not saturated yet as the incremental performance gain per % of improvement

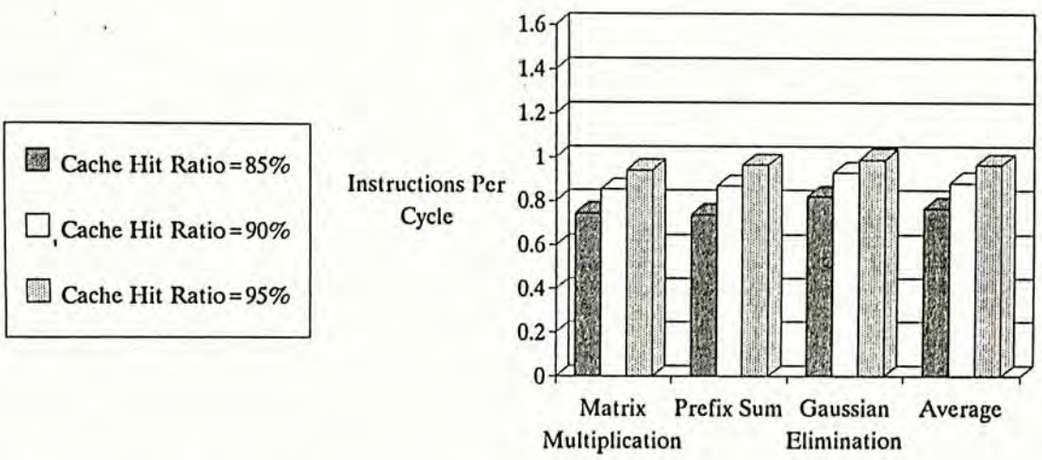
[10] One should be reminded that if only one instruction is fetched and decoded in each cycle, the maximum performance in terms of the average number of instructions completed per cycle cannot exceed one.

in the cache hit ratio is still increasing.

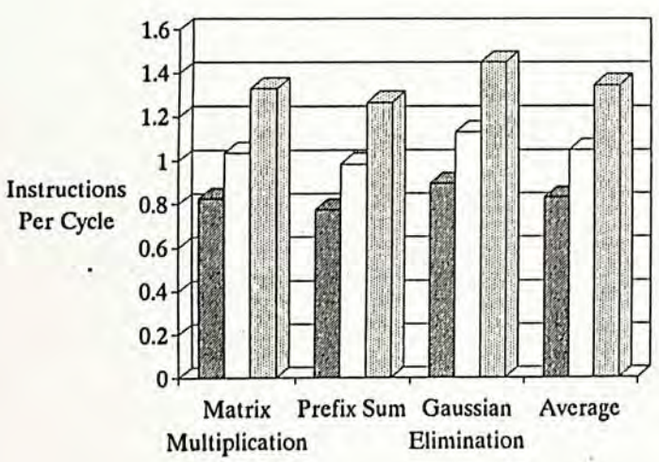
The result obtained is not surprising. If cache miss occurs frequently, the long latencies of the main memory accesses will on the one hand lengthen the total execution time. Therefore, the higher the proportion of memory accesses in the total instructions executed, the greater the performance gain that can be realized by improving the cache hit ratio.

Figure 5.23. The effects of varying the Cache Hit Ratio on Overall Performance

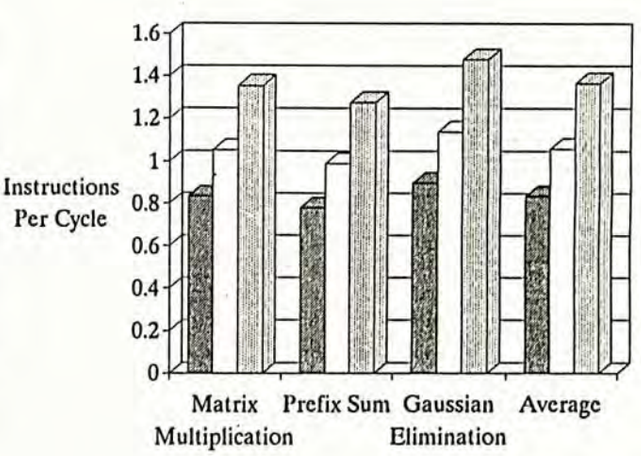
(a) Fetch Size = One Instruction Per Cycle



(b) Fetch Size = Two Instructions Per Cycle



(c) Fetch Size = Four Instructions Per Cycle



On the other hand, the unavailability of the required memory data/operand(s)

will block succeeding instructions from initiation (because of the Read-After-Write or true data dependency involved), thus further decreasing the overall efficiency[11]. The higher the degree of parallel execution as implied by increasing the fetch size, the greater the number of decoded/issued instructions which may be held from execution waiting for the completion of the main memory access. This explains the increasing significance of the cache's performance on the overall efficiency of the architecture.

5.7.2 Experiment 2

- *Objective* To demonstrate the benefits of adopting reservation stations.

- *Conditions* The basic configuration is assumed for the simulator. The number of memory load's reservation stations, reorder buffers (for memory stores) and the updating buffers are all fixed at eight. The fetch size is two instructions per cycle and the bandwidth of the common data bus is constrained to be two results per cycle only. So a maximum of two results can be delivered in each cycle. Finally, the maximum branch level is two. In other words, at most two conditional branch instructions can be outstanding at any time, and the third one will be held upon decoding.

Four cases are considered and the numbers of reservation stations for integer add (IAdd), integer multiply (IMul), floating-point add (FAdd) and floating-point multiply (FMul) are varied each time. In the first case, only one reservation station is used for each of IAdd, IMul, FAdd and FMul. In the second case, two reservation stations are available. Then four in the third case and finally, we have eight reservation stations for each type. With a 90% expected cache hit ratio, the incremental

[11] For cache misses involving store operations, the impacts are less severe as 'short-cut' paths from the Store Reservation Stations have been provided.

performance gain is determined.

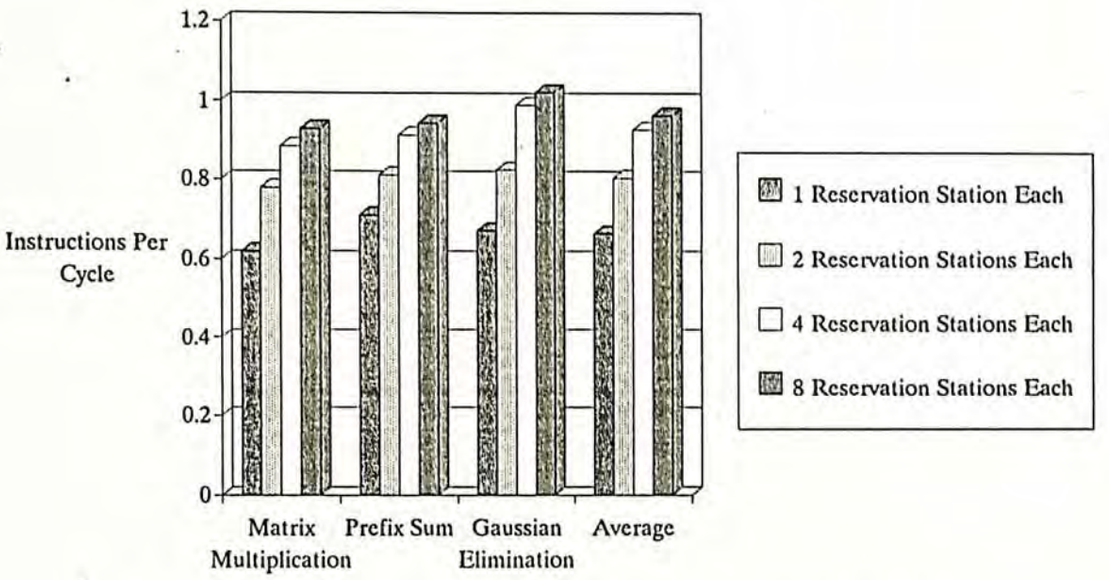


Figure 5.24. Performance attained by using 1, 2, 4, 8 Reservation Stations

- ***Results*** Figures 5.24 and 5.25 summarize the results obtained. Several observations should merit further discussions. First, on average, a 20% (20.856%) speedup in performance is realized by using two reservation stations instead of one. Please be noted that the case with only one reservation station used corresponds to the situation of having no reservation station at all. In other words, a significant performance improvement can be achieved at the price of adopting an extra single reservation station for buffering pending operation.

On the other hand, we can observe that the incremental performance gain diminishes as the numbers of reservation stations increase (the average incremental performance gain is 15.256% for four reservation stations, but drops sharply to 3.922% only when eight reservation stations are used). This can be expected since all of the fetch size, the bandwidth of the CDB and the maximum branch level are constrained to be two only. Together, they restrict the maximum parallelism to exploit and become the

limiting factors of performance improvement. The extra reservation stations added are probably left idle most of the time. Therefore, it is not worth the overhead incurred in hardware and control.

Intuitively, the optimal number of reservation stations for a particular function should relate in some way to its operation latency and the number of functional units available. As a rule, an operation with a longer latency should deserve more reservation stations because there would be a greater chance when ready-to-go operations are unnecessarily held (during decoding or pending for execution) because of the access conflict of the specific functional unit(s). On the other hand, the availability of more free functional units of the same type could help to ease such kind of 'congestion'. To conclude, although functions with higher frequencies of uses/executions should justify the allocation of more reservation stations, the fact that the particular mix of different functions in different applications can vary a lot. One should be well aware that considerable deviation of the actual performance from the expected one can be resulted because of this uncertainty.

	Number of Reservation Stations for IAdd, IMul, FAdd & FMul						
Benchmark	1 each	2 each	Increment (%)	4 each	Increment (%)	8 each	Increment (%)
Matrix Multiplication (Cache Hit %)	0.619127 (89.988%)	0.778765 (90.039%)	0.159638 (25.784%)	0.883702 (90.016%)	0.104937 (13.475%)	0.929869 (89.980%)	0.046167 (5.224%)
Prefix Sum (Cache Hit %)	0.711461 (89.996%)	0.810627 (89.950%)	0.099166 (13.938%)	0.913595 (90.041%)	0.102968 (12.702%)	0.943100 (90.000%)	0.029505 (3.230%)
Gaussian Elimination (Cache Hit %)	0.672066 (89.883%)	0.824442 (89.952%)	0.152376 (22.673%)	0.989477 (89.892%)	0.165035 (20.018%)	1.021561 (89.952%)	0.032084 (3.243%)
Average :	0.665382	0.804152	0.138769 (20.856%)	0.926837	0.122685 (15.256%)	0.963191	0.036354 (3.922%)

Figure 5.25. Simulation results obtained in Experiment 2

5.7.3 Experiment 3

- ***Objective*** To study the effects of varying the Fetch Size on the overall performance of the T-Architecture.

- ***Conditions*** The simulator will adopt the basic configuration with the maximum branch level constrained to be one level only, and a expected cache hit ratio of 90%. Three cases will be considered. By assuming a different fetch size every time (one instruction per cycle in the first case, two in the second, and finally four in the third), the performance attained is measured for each of the three benchmarks. Then the whole experiment is repeated, but now the bandwidth of the common data bus is increased to be capable of carrying at most two data per cycle. Results obtained are compared against those collected in the first situation for evaluation.

- ***Results*** Figures 5.26(a) and 5.27(a) depict the measured performance when the bandwidth of the CDB is limited to one result per cycle only. An average speedup of 14.575% can be achieved by fetching and decoding two instructions per cycle instead of one. Similar to the situation encountered in experiment 1, we suffer from the rule of 'diminishing return' again when we attempt to increase the fetch size further to 4 instructions per cycle. The average incremental performance gain is 0.651% only.

Although the maximal parallelism to exploit can be increased by fetching more instructions per cycle, the peak performance is not attained without promoting other factors, such as the bandwidth of the CDB and the numbers of functional units adopted. We say that the influence of the fetch size is saturated and the performance is sustained. Let's illustrate this point by considering the second case where the CDB can now carry at most two results per cycle. The effect of the fetch size on performance has been summarized in figures 5.26(b) and 5.27(b).

	Fetch Size (Number of Instructions Per Cycle)				
Benchmark	1	2	Increment (%)	4	Increment (%)
Matrix Multiplication (Cache Hit %)	0.841544 (90.005%)	0.959484 (90.007%)	0.117940 (14.012%)	0.973658 (89.992%)	0.014174 (1.477%)
Prefix Sum (Cache Hit %)	0.826560 (90.091%)	0.943839 (90.031%)	0.117279 (13.189%)	0.947854 (90.040%)	0.004015 (0.425%)
Gaussian Elimination (Cache Hit %)	0.888058 (90.065%)	1.026560 (90.087%)	0.138502 (15.596%)	1.026805 (90.120%)	0.000245 (0.024%)
Average :	0.851262	0.975338	0.124076 (14.575%)	0.981687	0.006349 (0.651%)

(a) Bandwidth of CDB = 1 Result Per Cycle

	Fetch Size (Number of Instructions Per Cycle)				
Benchmark	1	2	Increment (%)	4	Increment (%)
Matrix Multiplication (Cache Hit %)	0.854089 (89.999%)	1.037205 (89.985%)	0.183116 (21.440%)	1.054458 (90.004%)	0.017253 (1.663%)
Prefix Sum (Cache Hit %)	0.870383 (90.029%)	0.985405 (90.084%)	0.115022 (13.215%)	0.99185 (89.984%)	0.006445 (0.654%)
Gaussian Elimination (Cache Hit %)	0.930648 (90.039%)	1.131451 (90.067%)	0.200803 (21.577%)	1.141822 (89.929%)	0.010371 (0.917%)
Average :	0.883840	1.047939	0.164099 (18.567)	1.059186	0.0112465 (1.073%)

(b) Bandwidth of CDB = 2 Results Per Cycle

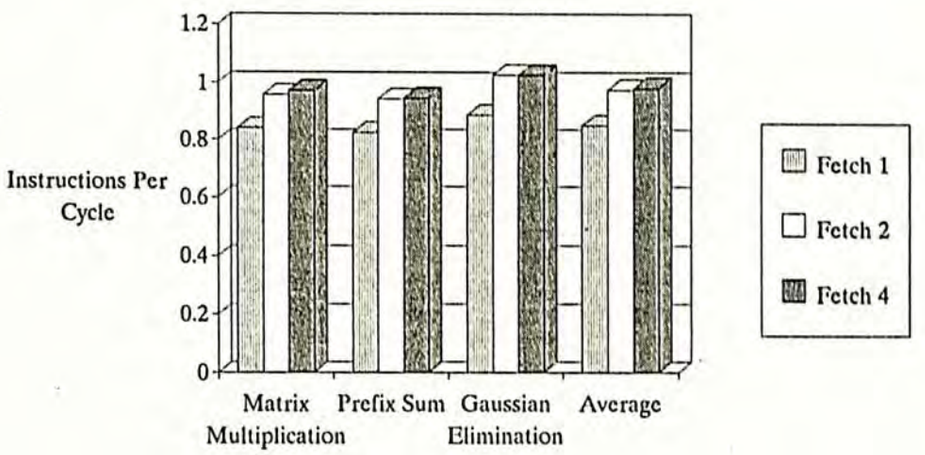
Figure 5.26. Simulation results obtained in Experiment 3

Now, 18.567% of speedup can be realized by doubling the fetch size from one instruction per cycle to two. More importantly, a further 1.073% of performance gain can be obtained by fetching four instructions per cycle. Thus we can see that the widening in the bandwidth of the CDB has on the one hand magnified the advantages of increasing fetch size and on the other hand, delayed its saturation.

Please be noted that we cannot boost the performance without limit by simply investing more and more computing resources (say by increasing the fetch size to 8 instructions per cycle). Sometimes, there may be constraints imposed by the way an algorithm is actually coded. For example, in the inner loop of the Gaussian elimination (see figure 5.21b), the procedural dependency created by the branch instruction can in

fact be overridden. By making use of the computational independence among different iterations of the loop, rewarding performance gain can be obtained by simple loop-unrolling. Similar techniques are numerous at both hardware and software levels. For example, register renaming has been proved to be an efficient remedy in dealing with output dependencies.

(a) Bandwidth of the CDB = One Result Per Cycle



(b) Bandwidth of the CDB = Two Results Per Cycle

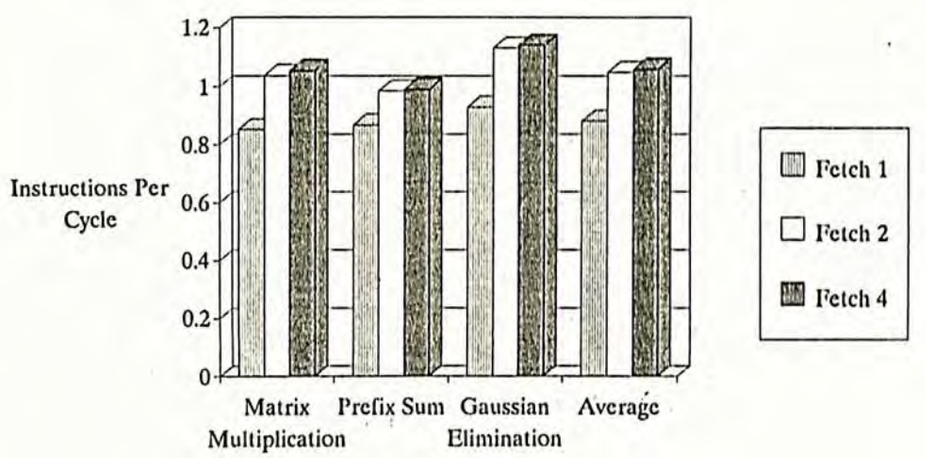


Figure 5.27. The impacts of varying the Fetch Size on the Overall Performance

On the other hand, the situation is somewhat different for the true limit implied by data dependency. Significant performance speedup can only be achieved by re-structuring the algorithm. Take for instance, in calculating the prefix sum of an array

(see figure 5.21c), as register F0 is used to accumulate the sum, the ADD instruction in each iteration of the loop cannot be fired until the completion of the corresponding ADD in the prior iteration. Similar restriction exists in the innermost loop of the matrix multiplication (see figure 5.21a) when the dot product of a row vector of the matrix A and a column vector of the matrix B is calculated.

Fortunately, efficient data parallel algorithms have been designed for various problems [Hillis&Steele86]. What we have to do is to work within the maximal parallelism allowed by a particular algorithm, and try to avoid the unnecessary burdens to performance in our architecture other than those imposed by data dependency.

5.7.4 Experiment 4

- ***Objective*** To study the bottleneck in performance implied by the bandwidth of the common data bus

- ***Conditions*** The simulator is configured as in experiment 3. Every time, a different bandwidth is assumed for the common data bus and the performance is evaluated for three different fetch sizes - one, two and four instruction(s) per cycle. The average number of instructions completed per cycle is measured in each case and more importantly, we want to monitor the traffic of the common data bus.

- ***Results*** It is natural to expect that performance can be promoted by increasing the bandwidth of the common data bus. The reason is that a larger capacity of the CDB could guarantee that computation results produced by various functional units can be forwarded to each of its destinations timely. This helps to save the performance loss incurred by the unnecessary postponement of the firings of operations which

are caused when one or more of the source operands are held in the result queue competing for the CDB. The simulation results summarized in figures 5.28 and 5.29 should help to justify our rationale.

Figure 5.28. Simulation results obtained in Experiment 4

	Bandwidth of CDB (Number of Results Per Cycle)				
Benchmark	1	2	Increment (%)	3	Increment (%)
Matrix Multiplication (Cache Hit %)	0.841544 (90.005%)	0.854089 (89.999%)	0.012545 (1.491%)	0.854500 (90.000%)	0.000411 (0.048%)
Prefix Sum (Cache Hit %)	0.826560 (90.091%)	0.870383 (90.029%)	0.043823 (5.302%)	0.871529 (89.874%)	0.001146 (0.132%)
Gaussian Elimination (Cache Hit %)	0.888058 (90.065%)	0.930648 (90.039%)	0.042590 (4.796%)	0.933086 (90.068%)	0.002438 (0.262%)
Average :	0.851262	0.883840	0.032578 (3.827%)	0.885113	0.001273 (0.144%)

(a) Fetch Size = One Instruction Per Cycle

	Bandwidth of CDB (Number of Results Per Cycle)				
Benchmark	1	2	Increment (%)	3	Increment (%)
Matrix Multiplication (Cache Hit %)	0.959484 (90.007%)	1.037205 (89.985%)	0.077721 (8.100%)	1.040185 (89.991%)	0.002980 (0.287%)
Prefix Sum (Cache Hit %)	0.943839 (90.031%)	0.985405 (90.084%)	0.041566 (4.410%)	0.985701 (90.091%)	0.000296 (0.030%)
Gaussian Elimination (Cache Hit %)	1.026560 (90.087%)	1.131451 (90.067%)	0.104891 (10.218%)	1.138691 (89.978%)	0.007240 (0.640%)
Average :	0.975338	1.047939	0.072601 (7.444%)	1.051128	0.003189 (0.304%)

(b) Fetch Size = Two Instructions Per Cycle

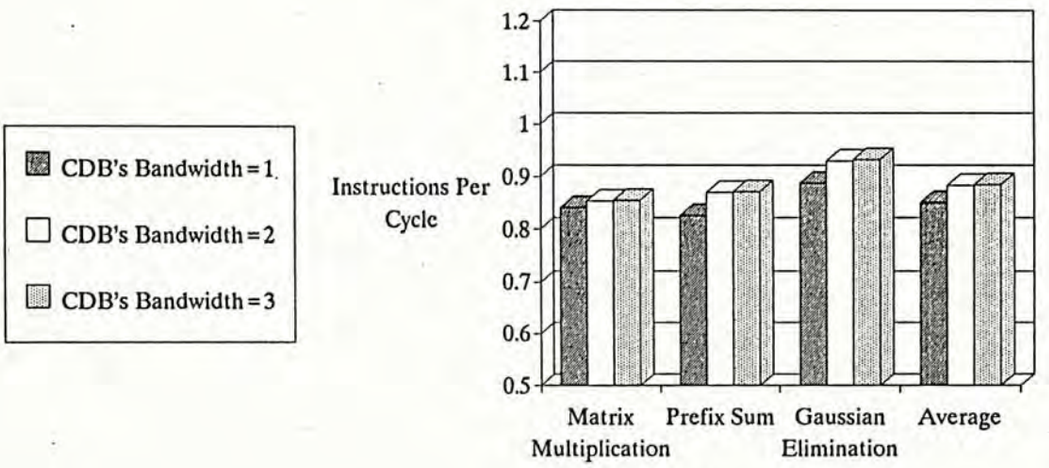
	Bandwidth of CDB (Number of Results Per Cycle)				
Benchmark	1	2	Increment (%)	3	Increment (%)
Matrix Multiplication (Cache Hit %)	0.973658 (89.992%)	1.054458 (90.004%)	0.080800 (8.299%)	1.061081 (89.974%)	0.006623 (0.628%)
Prefix Sum (Cache Hit %)	0.947854 (90.040%)	0.99185 (89,984%)	0.043996 (4.642%)	0.992278 (89.999%)	0.000428 (0.043%)
Gaussian Elimination (Cache Hit %)	1.026805 (90.120%)	1.141822 (89.929%)	0.115017 (11.201%)	1.157499 (90.031%)	0.015677 (1.373%)
Average :	0.981687	1.059186	0.077499 (7.894%)	1.066042	0.006856 (0.647%)

(c) Fetch Size = Four Instructions Per Cycle

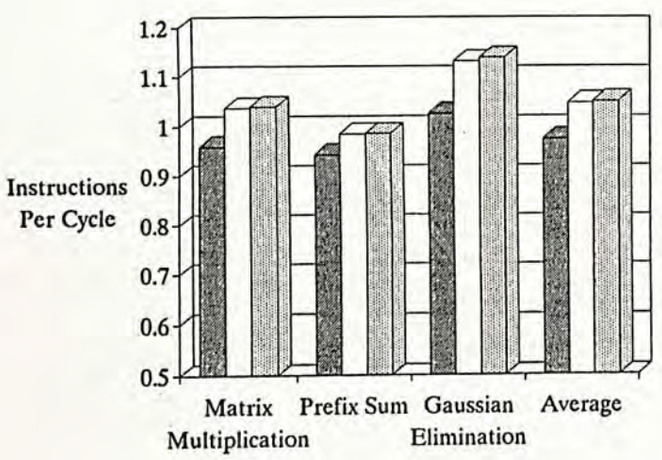
Ever since the common data bus was proposed in the original design of the IBM 360/91, it has been criticized of its narrow bandwidth - only a single piece of data can be carried in each cycle. When multiple functional units are adopted and that they are allowed to operate in an overlapped manner, more than one result can be produced in a single cycle, thus overloading the CDB.

Figure 5.29. The effects of the Bandwidth of the CDB on Overall Performance

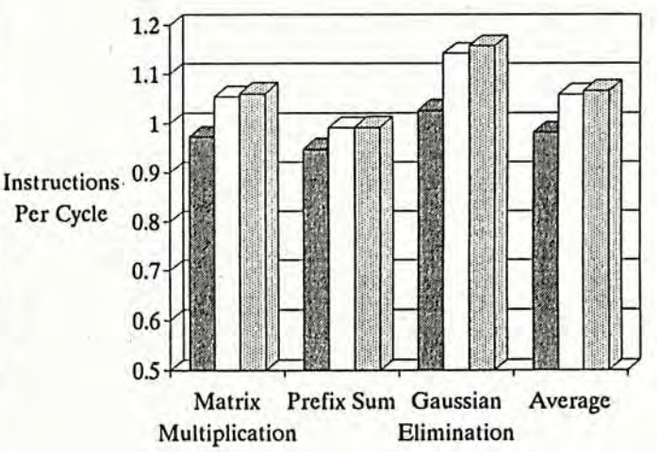
(a) Fetch Size = One Instruction Per Cycle



(b) Fetch Size = Two Instructions Per Cycle



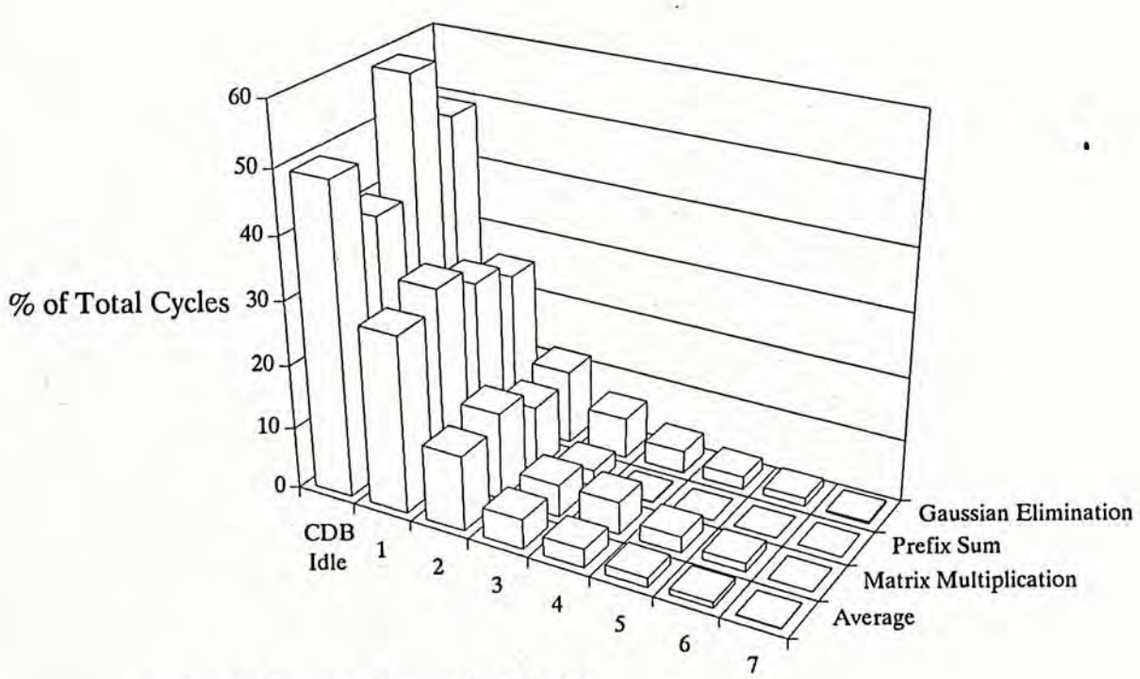
(c) Fetch Size = Four Instructions Per Cycle



The situation becomes even worse as the degree of parallel execution promotes as a consequence of the increase in fetch size. Operations are fired and completed as

early as possible so as the generations of outputs. We say that 'multiple issue implies multiple completion'. In our experiment, this inadequacy of the bandwidth of the CDB has accounted for 3.834% to 8.002% of performance loss, and that the average CDB-queueing time for each result ranges from 0.787 cycles to 1.515 cycles[12].

At first glance, we may soon arrive at the conclusion to save the performance loss by simply doubling the bandwidth of the CDB. However, a careful re-thinking of the experimental result should reveal that the large overhead which can be incurred in hardware should not be a good bargain for only 8% increase in performance. To tackle the problem, two observations may be useful.



	Number of Results Competing for the CDB (% of Total Cycles)							
Benchmark	CDB Idle	1	2	3	4	5	6	7
Matrix Multiplication	40.419%	31.291%	14.083%	5.178%	5.156%	2.570%	1.303%	0%
Prefix Sum	58.672%	28.145%	10.458%	2.546%	0.178%	0%	0%	0%
Gaussian Elimination	49.254%	25.340%	11.732%	6.481%	3.616%	2.009%	1.241%	0.326%
Average :	49.449%	28.259%	12.091%	4.735%	2.984%	1.526%	0.848%	0.109%

[12] The average performance loss for various fetch sizes of one, two and four instructions per cycle are 3.834%, 7.284% and 8.002% respectively, and that their corresponding average CDB-queueing times are 0.787 cycles, 1.366cycles and 1.515 cycles.

Figure 5.30. Studying the Actual Loading of the Common Data Bus

(Fetch Size = 1 Instruction Per Cycle, Bandwidth of the CDB = 1 Result Per Cycle, Cache Hit Ratio = 90%)

First, let's consider the actual loading of the common data bus. Assuming that the fetch size is one instruction per cycle, the bandwidth of the CDB is 1 result per cycle and the expected cache hit ratio is 90%, we obtain the simulation results depicted in figure 5.30.

As shown, the common data bus is capable of servicing all outputs in the result queue in more than 75% of the time. In the remaining 25% of the total cycles, as many as 7 results can be competing for the CDB in the same cycle. Surprising enough, the common data bus is in fact idle in nearly 50%[13] of the time. In other words, the CDB is functioning at half efficiency only and we have not fully utilize its available bandwidth.

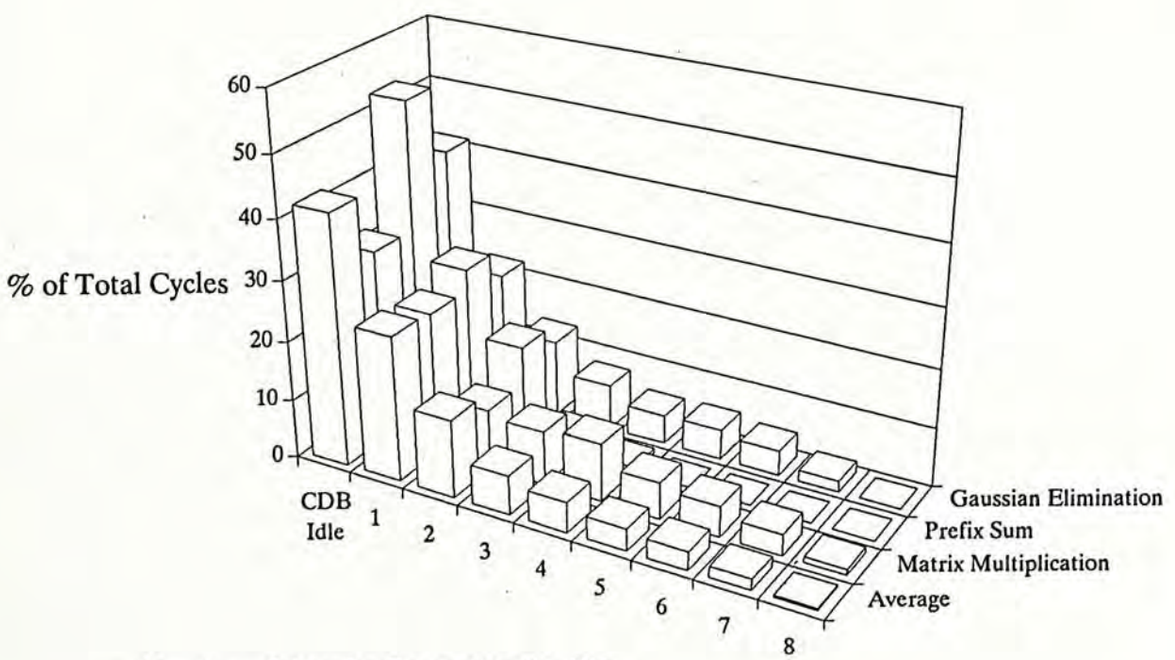


Figure 5.31. Actual Loading of the Common Data Bus

[13] The wastage of the bandwidth of the CDB is less severe with a small fetch size. With reference to figure 5.31, for a fetch size of two instructions per cycle, the common data bus is idle in only about 42% of the time. However, as many as 8 results can be competing for CDB in the same cycle.

(Fetch Size=2 Instructions Per Cycle, Bandwidth of the CDB=1 Result Per Cycle, Cache Hit Ratio=90%)

Therefore, performance gain can still be 'squeezed' out from the wasted bandwidth of the CDB. A possible strategy to increase its utilization level is by moving instructions back and forth with consideration of their relative time of completion. Instruction scheduling of this type is commonly implemented at the software level.

Still another insight concerns the actual number of forwarding destinations of each result. With reference to figure 5.32, if only one instruction is fetched and decoded in each cycle and that the bandwidth of the CDB is kept at one result per cycle, we can see that the majority of results (about 79.356%) are forwarded to a single destination only. In addition, the average number of forwarding destinations for each result is 1.267 (similar phenomenon has been reported in [Johnson91]).

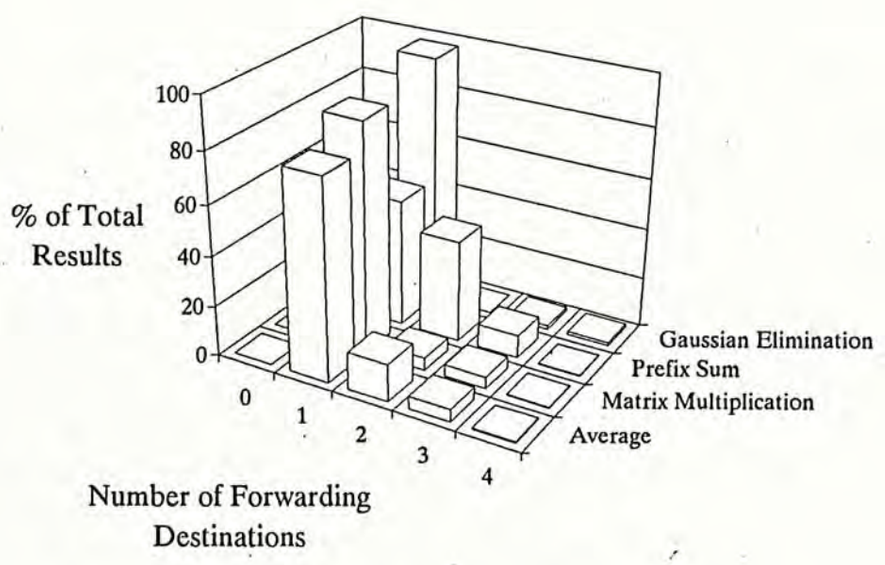


Figure 5.32. Studying the Number of Forwarding Destinations of Results Carried by CDB

(Fetch Size=1 Instruction Per Cycle, Bandwidth of the CDB=1 Result Per Cycle, Cache Hit Ratio=90%)

Doubtless, the idea of the common data bus helps to realize the concept of procedure graph optimization. The need to forward a single piece of data to multiple destinations is urged by the uses of Serial-to-Parallel Transformations (SP) in which

relay data transfers are being replaced by simultaneous broadcasting. However, as the available inherent parallelism to exploit is limited by a small fetch size (and it can be expected that less SPs can be achieved as a result), the common data bus may sound less attractive. In fact, we have found that when the fetch size is increased to two instructions per cycle, the average number of forwarding destinations will become 1.519 and as many as 28.296% of the total results need to be forwarded to more than one destination (see figure 5.33).

	Number of Forwarding Destinations (% of Total Results)					
Benchmark	0	1	2	3	4	Average
Matrix Multiplication	0.0732%	87.7793%	0.9015%	2.4682%	8.7777%	1.3210
Prefix Sum	0.0007%	38.5357%	45.8807%	1.1151%	14.4679%	1.9151
Gaussian Elimination	0.0003%	88.7214%	0.1143%	1.4546%	9.7093%	1.3215
Average :	0.0248%	71.6788%	15.6322%	1.6793%	10.9849%	1.5192

Figure 5.33. Studying the Number of Forwarding Destinations of Results Carried by CDB

(Fetch Size = 2 Instructions Per Cycle, Bandwidth of the CDB = 1 Result Per Cycle, Cache Hit Ratio = 90%)

The Direct Tag Search method (DTS) first proposed in [Weiss&Smith84] has made use of this characteristic. In an attempt to simplify the hardware and eliminate most associative matching of tags involved in forwarding, the DTS imposes the restriction that a particular tag can be referred only once. A second attempt to use it will be held upon decoding. In other words, there can be at most one recipient for each computation result (produced by a functional unit). Its address will be associated with the tag concerned and forwarding is replaced by a direct tag search mechanism implemented with a table indexed by tags. Simulations have revealed that performance will not be affected much.

But that should not be the whole story. First, as we have just mentioned, the average number of forwarding destinations will increase as more instructions are fetched in each cycle (compare figures 5.32 and 5.33). The fact that a result can be forwarded to one destination only will degrade performance much then. In particular,

Serial-to-Parallel Transformations of the type shown in figure 5.34 cannot be achieved now. When more instructions are considered at the same time, optimizations similar to this will be numerous. The cost of missing these opportunities can be foreseen.

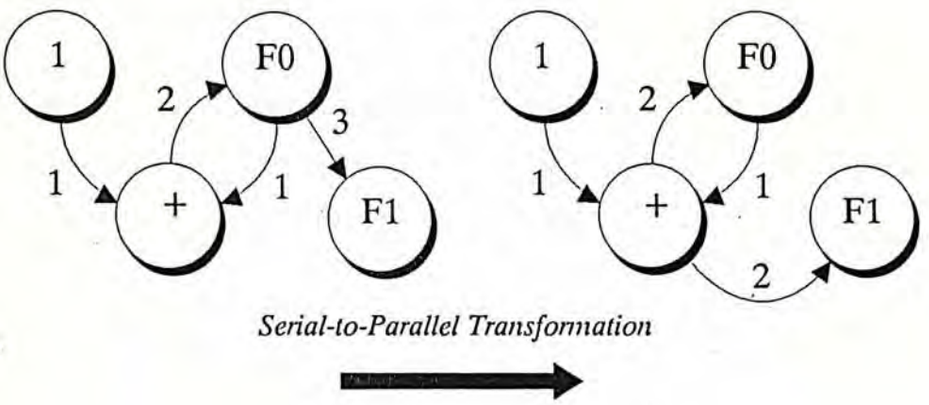


Figure 5.34. SPs of this kind become prohibited when DTS is used

Any attempt to achieve the distribution of a single result to multiple destinations via a mechanism other than that adopted by the common data bus (e.g. by keeping a list of destination addresses instead of one) will on the one hand suffer from the hardware overhead pointed out in [Johnson91] (e.g. list searching), and on the other hand, Store-Store Cancellations (SSC) are now achieved at a much higher cost. As illustrated by the example in figure 5.35 where two instructions are considered in turn :

```
MUL  F0,@1
LD   F0,A[1]
```

When the second instruction LD is decoded, an entry will be added to the list associated with the tag 22 (of the Load reservation station) storing the identity/address of the register F0 so that when the data at A[1] arrives upon completion of the memory load operation, the content of F0 will be updated correctly (via Direct Tag Search). At the same time, its old entry should be removed from the list associated with the tag 63 (of the Floating-point Multiply reservation station). Otherwise, if it happens that the multiplication completes after the memory load, the content of F0 will be modified

incorrectly. Thus we can see that each application of SSC involves at least two modifications (and searches) of the Tag Table.

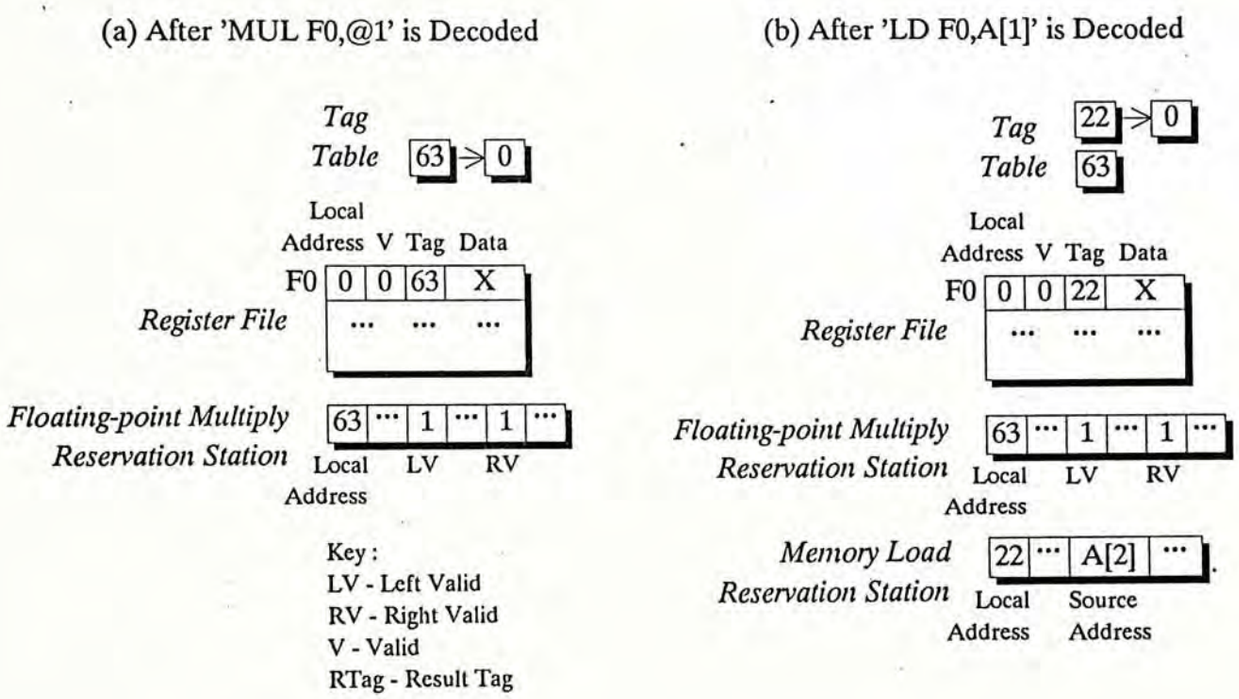


Figure 5.35. The application of an SSC will involve searching and modifying the Tag Table if Direct Tag Search is used

It is not the maximum number of prospective destinations of a data (that is, the number of times a result tag can be referred) that distinguishes the DTS method from forwarding via Common Data Bus. The fundamental difference is that the DTS method uses forward pointers instead of backward pointers to represent a data transfer (arc). In particular, we require a data to specify its list of destinations instead of allowing possible destinations to specify what data they want.

But as we have mentioned, while the source of a data is unique, its destination is usually not. This asymmetry leads to a single consequence - no matter how long the address list for each entry of the Tag Table can be, there is always a possibility that we will run out of its maximum capacity (theoretically, CDB corresponds to a DTS with address lists of infinite length). And more importantly, the hardware overhead and execution inefficiencies incurred by the representation and manipulation of the address

list will increase rapidly with its length, effectively making the DTS method unfavorable as a substitute for the Common Data Bus (completely).

Chapter 6 Predictive Procedure Graph Optimization in The S-Prototype

As highlighted in the last chapter, the success of the T-Architecture, though exemplifies certain advantages of backward-pointer representation scheme, does exhibit some of its weaknesses. The fact that only a single tag can be associated with each node dictates the application of real-time optimization only. The belief that predictive optimization can reveal more fruitful results (by increasing the scope to apply optimizing graph transformations) has motivated us. Our efforts give rise to the S-Prototype ("S" stands for "Superscalar").

Originally evolved as a superscalar design based on the T-Architecture, the S-Prototype resembles the T-Architecture in many aspects, e.g. the design of the memory system, the use of backward pointers for representing procedure graphs, the use of speculative execution, etc. Yet, several features make the S-Prototype more unique and effective.

While a "simple" tag similar to the one adopted in the T-Architecture is still associated with each node (each storage location such as a register) to achieve real-time optimization, multiple "timed tags" can now be associated with each node which are centralized in the Multitag Pool. By fetching and decoding multiple instructions every cycle, an algorithm under consideration manifesting itself as a global procedure graph will be represented by a set of timed tags. Their manipulations realize limited predictive optimization[1].

On the other hand, the fact that the Multitag Pool is basically maintained as a doubly-threaded list provides more global information of the procedure graph under

[1] The fact that we are still constrained to examine the instruction stream sequentially has limited the effectiveness achieved, although it is done ahead (predictive) of real-time execution.

consideration, effectively offering access convenience not supported by the simple (and single) tagged T-Architecture. In succeeding discussions, we will consider them in detail. Again, a simulation program has also been written for the S-Prototype.

6.1 Keys to Higher Performance

Ever since the invention of computers, scientists have invested their every effort to speedup the operations of computers. Now we have the common beliefs that duplicating computing resources can lead to folds of speedup of several fold. The advances in VLSI technology have turned many of our dreams into reality. Now more than one million transistors can be packed into a single chip. But that should not be the whole story.

We claim that the full performance potential has not been fully explored. While this issue is somehow application-dependent, the inefficiencies of conventional programming languages, compilers as well as prevailing architecture philosophies and execution semantics should be blamed for. Doubtless, there is still a long way to go. There is no unique promising approach. But we now have the common consent that performance can be boosted by (see [Johnson91], [Hennessy&Patterson90] and [Keller75]) :

- issuing multiple instructions per cycle
- issuing and executing instructions out-of-order, thus effectively overriding the unnecessary sequencing of instructions caused by the conventional sequential programming languages
- overlapping the execution of different instructions, e.g. pipelining (the lower the level of overlapping, the more rewarding the result achieved)

6.2 The Superscalar Approach

The performance potential (usually referred as the machine parallelism [Johnson91]) of a uniprocessor equipped with multiple resources (decoders, ALUs, etc) comes from the ability to execute more than one instruction in parallel. Pipelining explores this idea by overlapping the different processing stages of different instructions. Effectively, the number of instructions executed per cycle can be significantly increased. RISCs try to shorten the cycle time by simplifying instruction formats, at the expense of increasing the number of instructions by, hopefully, a tolerable amount.

Superscalar processors [Johnson91] approach the problem by making use of instruction-level parallelism. Taking one step further, superscalar techniques allow the issue of multiple RISC-like independent instructions per cycle by real-time resolution of causality and precedence constraints. At the same time, pipelining can be adopted at various levels (e.g. pipelined instruction processing cycle and pipelined functional units). The overall result is a much wider pipeline.

Although there is no reason to exclude the use of software optimization techniques, superscalar processors rely mainly on dynamic scheduling at the hardware level, thus releasing much workload of the compilers. And that's why superscalar solutions (and processors) are usually credited for their compatibility and predictable performance across a wide range of applications (even unscheduled code can be executed quite efficiently).

6.3 Processor Architecture of the S-Prototype

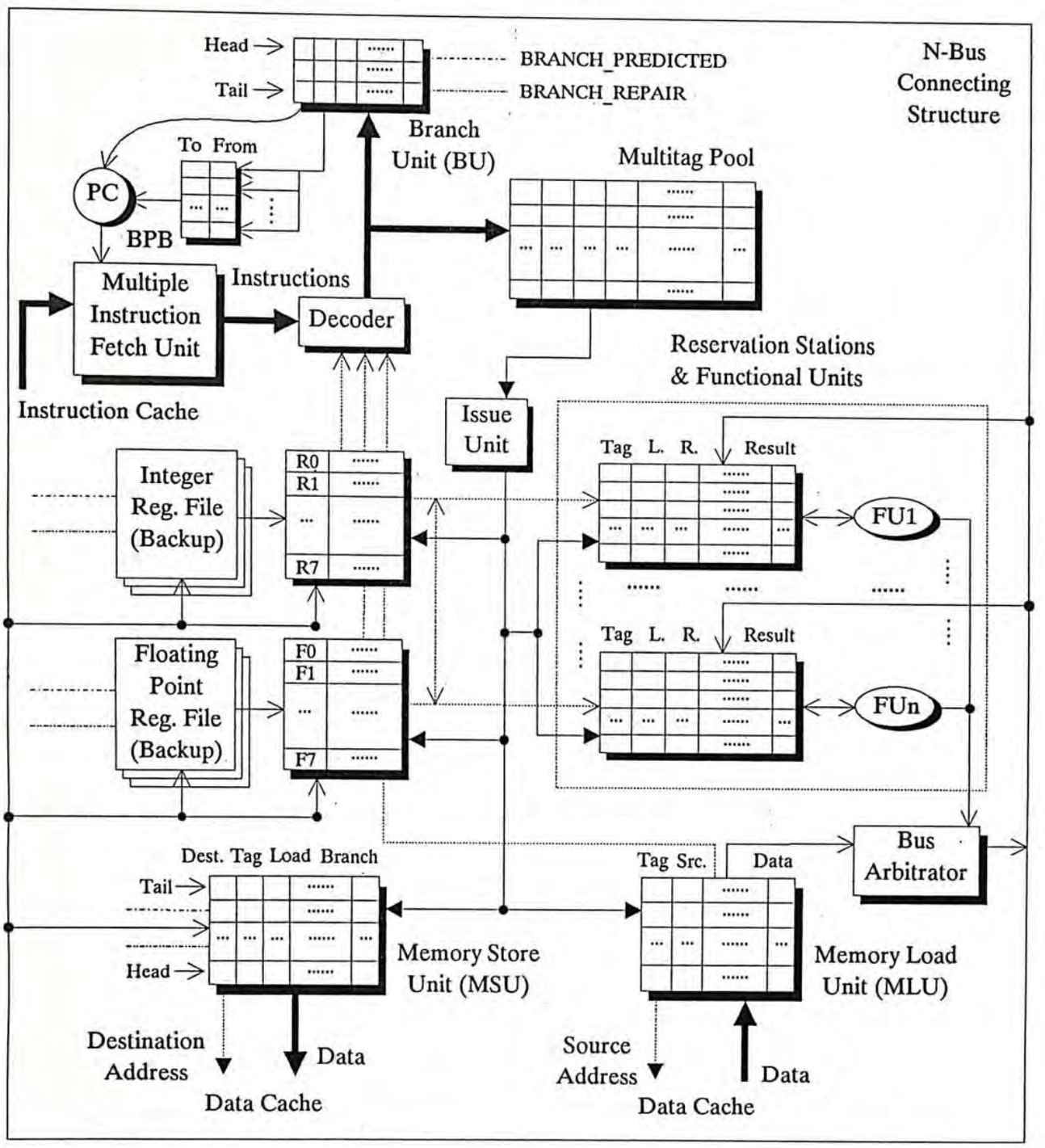
Figure 6.1 depicts the block diagram of the S-Prototype Processor. In each cycle, multiple instructions are fetched by the Multiple Instruction Fetch Unit. Simple branch prediction is achieved by the Branch Unit with the assist of the Branch Predicting Buffer (BPB).

Upon decoding, the underlying data transfers of instructions are represented using timed tags in the Multitag Pool. Effectively, a procedure graph is manifested. Two transformation rules - Store-Store Cancellations and Serial-to-Parallel transformations have been implemented to achieve optimization. Functionally, the Multitag Pool resembles the scoreboard [Hennessy&Patterson90] where causality constraints are analyzed and unnecessary sequencing is removed. Data transfers (or operations) with all static dependencies resolved are gathered by the Issue Unit. They will be dispatched to the respective reservation stations via the Issue Bus.

Reservation Stations serve as buffers where pending operations wait for their dynamic dependencies to be cleared, firing then occurs autonomously. As a result, control can be distributed and multi-threaded. In every cycle, a result together with its identification tag will be collected by the Bus Arbitrator from each finishing functional unit, which will then be carried via the N-Bus Connecting Structure. Extensive hardware tagging allows the efficient forwarding of computation results to multiple destinations.

To summarize, the S-Prototype optimizes performance in the following ways :

- Procedure graph optimization extracts parallelism at the data transfer level, thus allowing the maximum possible amount of overlapping
- Lookahead : Fetching and examining multiple instructions at the same time
- Out-of-order issue and execution of operations
- Expedite the issue of operations as soon as all true static dependencies are removed
- Cancel unnecessary operations (via Store-Store Cancellations) and remove relay data transfers (via Serial-to-Parallel Transformations)
- At the execution level, the control is multi-threaded. Dynamic dependencies are resolved by extensive tagging and forwarding



Key :

BPB Branch Predicting Buffer
Issue Operations (Issue Bus)
Tag or Data
Tag and Data
BRANCH_PREDICTED
BRANCH_REPAIR

Figure 6.1. The Architecture of the S-Prototype

6.4 Design Strategies of the S-Prototype

To fully enjoy the benefits of superscalar design, a couple of things should be handled properly and the design of the S-Prototype represents a series of compromises among

the alternatives. The ultimate goal is, given the hardware resources equipped, to exploit the maximum amount of parallelism available in general-purpose applications.

6.4.1 Fetching Multiple Instructions

First, there must be enough instruction-level parallelism to feed the wide bandwidth of the superscalar pipeline. While this issue is mostly application-dependent [Smith et al.89], an efficient design should be capable of extracting the maximum amount of parallelism available.

Upon this issue, two factors are significant. First, no matter what parallelizing algorithm is used, there is no denying that the larger the scope to examine, the more effective the result obtained. So we need a way to fetch multiple instructions at the same time. A block of instructions being fetched as a whole effectively defines a sub-algorithm of the total computation to optimize. In the S-Prototype, the fetch unit as well as the decode unit are duplicated. In each cycle, a maximum of 4 instructions are fetched from the dynamic instruction stream and decoded.

6.4.2 Handling Procedural Dependencies : Branching Instructions

However, the presence of procedural dependencies (conditional and unconditional branch instructions) will corrupt the normal sequential fetch sequence and thus should be resolved properly so that the fetch efficiency (the number of instructions fetched per cycle) will not be sacrificed much. Other factors such as instruction alignment are also significant. Interested readers should refer to [Smith et al.89] for a detailed discussion.

We can choose to freeze the fetch unit every time a conditional branch instruction is encountered until the corresponding procedural dependency is resolved. But then we will lose much performance. So instead, the S-Prototype has adopted a

simple branch prediction mechanism to allow the execution to continue in the predicted path in a conditional mode.

6.4.2.1 Branch Unit and Branch Predicting Buffer

In the S-Prototype, we have chosen not to design condition code. Instead, a conditional branches on the value of a specific (floating-point or integer) register (zero, positive or negative). As a result, branch instructions assume the following formats :

Format	Meanings	
BZ R_i , Address_If_Taken	PC ← Address_If_Taken	if (R_i) = 0
BG R_i , Address_If_Taken	PC ← Address_If_Taken	if (R_i) > 0
BL R_i , Address_If_Taken	PC ← Address_If_Taken	if (R_i) < 0

With reference to figure 6.1, each branch instruction encountered is stored in the Branch Predicting Buffer with the destination address it has last taken, which is used as the predicted new PC every time when the same branch instruction is fetched. If no entry exists (e.g. when the branch is first come across), Address_If_Taken is assumed. Execution continues conditionally while the branch instruction is being handled by the Branch Unit.

The Branch Unit has 3 entries, effectively allowing nested branching to a maximum depth of 3 levels. This implies that at any time there can be up to 3 outstanding conditional branch instructions. The evaluation of branch instructions are totally ordered. In every cycle, the first buffered branch instruction is examined, and execution will be initiated if its data dependency has been removed, that is, the content of the specific register of interest is valid.

When a branch instruction completes, the actual outcome (the new PC) can be compared with the early prediction. A successful guess will generate a

BRANCH_PREDICTED signal. Otherwise, a BRANCH_REPAIR signal will be sent out. Either signal will be broadcasted system-wide, and components concerned will react accordingly. Generally speaking, upon receipt of a BRANCH_PREDICTED signal, we will have the temporary computation results formally committed and useless backup information discarded. The case of BRANCH_REPAIR is a little complicated. First, instruction fetch will be suspended. Then, temporary results are thrown away and the correct machine state is restored using backup information. Finally, instruction fetch resumes at the new correct PC.

6.4.2.2 Branch Repairing - Recovering Machine State

To prepare for a branch misprediction, we need a mechanism to repair the original state of the machine before we restart from the correct control path. A hybrid scheme has been adopted - memory and registers are handled differently.

Checkpoint repair [Hwu&Patt87] is adopted for register data. A number of backup states are kept for the integer registers and the floating-point registers respectively, which is equal to the active depth of branch nesting. As an example, consider the following code sequence :

```
        MOV   R3,@-3
LOOP:   IADD  R3,@1
BRANCH: BL    R3,LOOP
        MOV   R2,R3
```

As shown in figure 6.2, the dynamic execution of the loop has resulted in three outstanding unresolved branch instructions $Branch_1$, $Branch_2$ and $Branch_3$. Each backup state, $Backup_1$, $Backup_2$ and $Backup_3$ serves as the checkpoint w.r.t the corresponding branch instruction. For example, $Backup_1$ maintains the state of the integer register file before the instruction $Branch_1$ (R3=-2). In other words, the effects

of all instructions coming after $Branch_1$ will not be reflected in $Backup_1$. With out-of-order execution and lookahead, the register files always maintain the lookahead state of the machine.

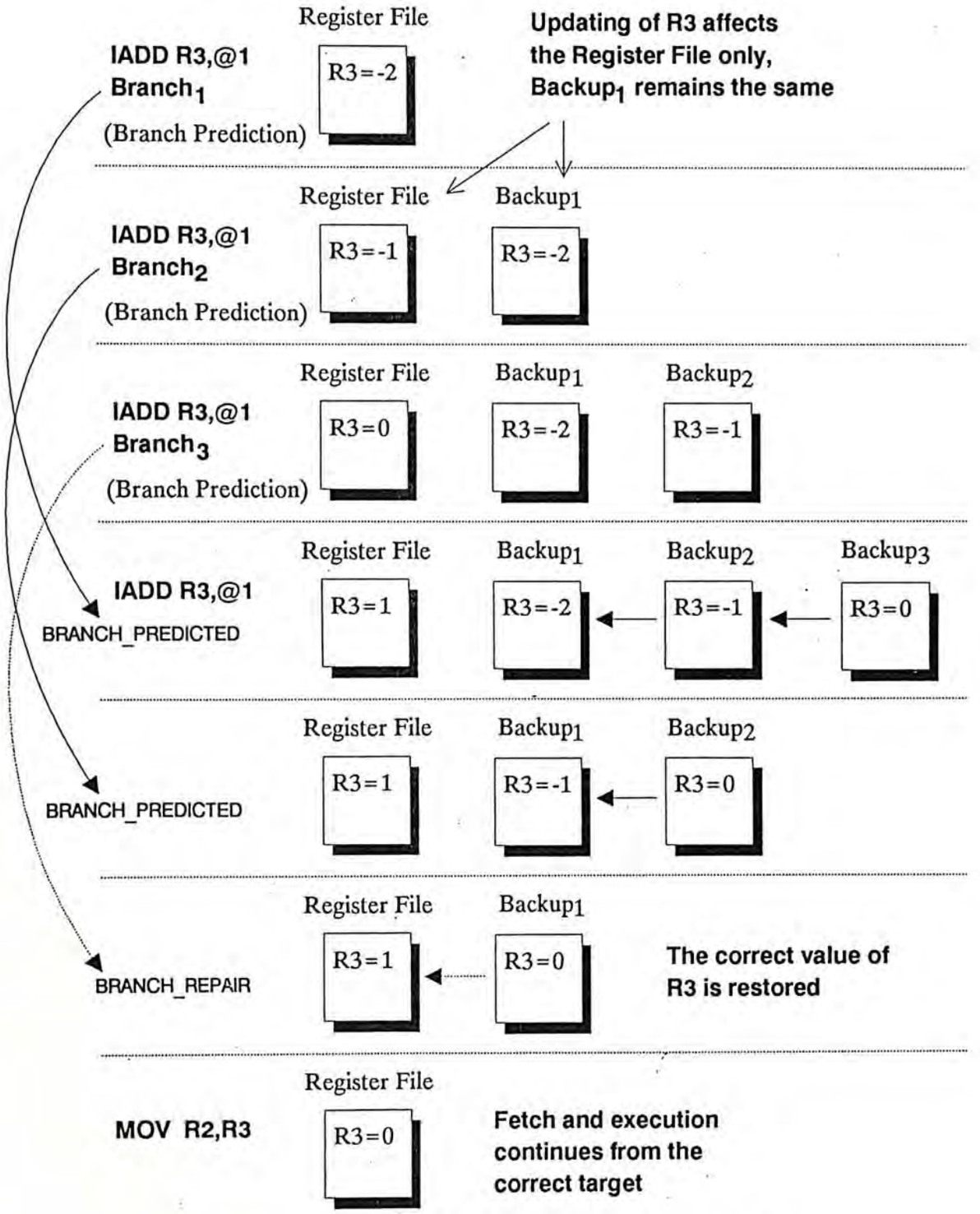


Figure 6.2. An example of checkpoint repair of register data

The receipt of the first BRANCH_PREDICTED signal implies that the outermost branch instruction $Branch_1$ has been resolved. As a result, the contents of

$Backup_1$ are discarded. $Backup_2$ will become the new $Backup_1$ and $Backup_3$ will replace $Backup_2$. Similar argument applies for $Branch_2$ upon receipt of the second BRANCH_PREDICTED signal.

However, the situation is different for $Branch_3$ where we have a misprediction. A BRANCH_REPAIR signal will be received and the states of the integer and floating-point register files have to be restored from the respective current $Backup_1$ (the correct value of R3 should be 0 indeed) while the contents of $Backup_2$ and $Backup_3$ will be simply discarded. At the same time, data transfers of lookahead instructions which are dependent on this mispredicted branch and which have not yet been issued will be eliminated from the Multitag Pool. Execution then resumes at the correct target as depicted.

However, checkpoint repair is not feasible for memory data because of its large volume. We need an alternative - a reorder buffer similar to the one proposed in [Smith&Pleszkun88]. With reference to figure 6.1, the Memory Store Unit (MSU) of the S-Prototype is implemented as a first-in-first-out queue and all memory store operations are totally ordered.

Each entry in the MSU maintains a field Branch which is initialized by the current depth of branch-nesting when the corresponding (branch) instruction is decoded. Every time when a BRANCH_PREDICTED signal is received, the branch field of each active entry will be decremented by one. In every cycle the first entry in the MSU will be examined. If the data to be stored is ready and the value of the branch field is zero, the store operations can be fired. Otherwise, this operation as well as all succeeding ones will be held until the respective data and/or procedural dependencies are resolved. On the other hand, when a BRANCH_REPAIR signal is received, all MSU entries with a nonzero Branch field will be flushed out.

Please be noted that in the above discussions, we have not ruled out the possibility of using software scheduling techniques, such as delayed branching, to assist the resolution of procedural dependencies. But the point is that as the instruction fetch bandwidth is much wider in superscalar machines, more sophisticated algorithm should be called for in order to schedule enough independent instructions to fill in a delay slot. For example, a machine with a fetch bandwidth of 4 instructions and a branch delay of 2 cycles would demand 8 independent instructions. Common applications (especially non-vectorizable ones) normally cannot afford [Johnson91].

6.4.3 Extensive Tagging and Result Forwarding

The success of the procedure graph approach and the S-Prototype in achieving optimization relies on a special mechanism - namely, the ability to distribute a piece of data to more than one destination.

The extensive applications of the SP transformations effectively minimize relay data transfers by simultaneous broadcasting in the same cycle. This implies that a piece of data can have more than one recipient. A first intuition may reveal that we can keep a list of destination addresses with each piece of data and then travel to each place accordingly. But then the average delays of transfers will be lengthened. Even worse, the number of destinations can vary a lot, or we have to place an unreasonable limit on it.

Therefore, we need a different approach. The data distributor (e.g. the bus) and the data itself now play a more 'passive' role. A destination as the consumer of a piece of data will agree with its producer an identification for that data - a tag. All other consumers waiting for the data keeps the tag. When the data of interest is actually produced, it is placed on a Common Data Bus which is capable of reaching each possible destination, where simultaneous tag matching occurs. Those hardware components with a matching tag gate in a copy of the data.

This is precisely the approach first adopted by the IBM 360/91 [Tomasulo67]. The S-Prototype exploits this technique to the extreme. All entries in the Multitag Pool and the reservation stations are given names - tags. These serve as identifications of a particular result produced by a specific functional unit.

An innovation in the S-Prototype is to make use of tagging to achieve a special kind of synchronization - the ordering of memory accesses. To avoid inconsistency, a memory store should not proceed until any outstanding memory load(s) to the same address has/have been finished. By tagging the data loaded, the pending branch operation will have its dependency cleared upon seeing a matching tag on the N-bus (see section 6.4.7 also).

The extra hardware required includes the tag-matching components for the Multitag Pool and reservation stations. In addition, a N-bus Connecting Structure is implemented which serves a function similar to the Common Data Bus of the IBM 360/91, but it is capable of forwarding more one result in each cycle (N being re-configurable in our simulator).

6.4.4 Static and Dynamic Data Dependencies

Generally speaking, data dependencies can be static or dynamic [Wang&Wu91]. Static dependencies are predictable by examining the control and data flow of the program. For example, with reference to the following code sequence :

```
MOV  R0,@0
IADD R0,@3
```

the second instruction cannot be started before the initiation of the first. On the other hand, dynamic dependencies arise because of exceptional events. These address

such uncertainties as latencies of operations, memory access latency and procedural dependencies. As an example, consider the following case :

```
LD    R0,A[1]
IADD  R0,@3
```

Without the knowledge of the actual memory latency (which is different for a cache hit and a cache miss), we cannot tell exactly how many cycles after the issue of the Load instruction that the IADD operation can be safely launched.

The problem of resource conflicts is just another side of the same coin. Causally speaking, it represents another kind of dynamic dependency. An operation with all other dependencies resolved (for example, all its source operands have become ready) still has to wait for the particular functional unit to become free.

One feasible approach is to hold the issue of an operation/instruction until all of its dynamic and static dependencies are removed. But then the execution of succeeding ready-to-go instructions will be unnecessarily postponed.

Consequently, the S-Prototype adopts a different approach. Reservation stations are implemented and instructions/operations with all static data dependencies removed can be issued if there are free reservation stations, where they wait for the particular dynamic dependencies to be resolved.

6.4.4.1 Handling Static Dependencies by Using the Multitag Pool

Waiting for the static data dependency to be removed is a synchronous event. In our design, the Multitag Pool serves this purpose. Static data dependencies of data transfers are manifested by the relative magnitudes of the Time fields in their respective timed tags (please refer to section 6.4.5 for a detailed discussion).

Taking one step further, static data dependencies can be classified as : Read-After-Write dependency (RAW), Write-After-Read dependency (WAR) and Write-After-Write dependency (WAW) [Stone87]. The examples in figure 6.3 below illustrates the meaning of each of them.

Figure 6.3. The 3 types of static data dependencies

S1 : A = B + C S2 : D = A	S1 : A = B + C S2 : B = C + 2	S1 : A = B + C S2 : D = A + 2 S3 : A = 3
(a) Read-After-Write (RAW)	(b) Write-After-Read (WAR)	(c) Write-After-Write (WAW)

Note : In (a), the execution of S2 must follow S1, written as S1δS2, if D is to receive the correct result of B+C. In (b), the original content of the location B must be preserved until S1 has completed its read-access to it. Therefore, S1 should precede S2 in execution. Finally, in (c), S3 must follow S1 lest A and D will contain the wrong values. Note that the constraint will still hold even if S2 is absent.

The maximal parallelism should only be constrained by Read-After-Write dependency and precedence relations since they are unavoidable. In other words, Write-After-Read dependency and Write-After-Write dependency should be removed.

6.4.4.2 Handling Dynamic Dependencies by Using Reservation Stations

Depicted in figure 6.4 is an example illustrating how reservation stations help to resolve dynamic dependencies. The code sequence :

```
LD    R0,A[1]
IADD  R0,@3
IMUL  R0,@2
```

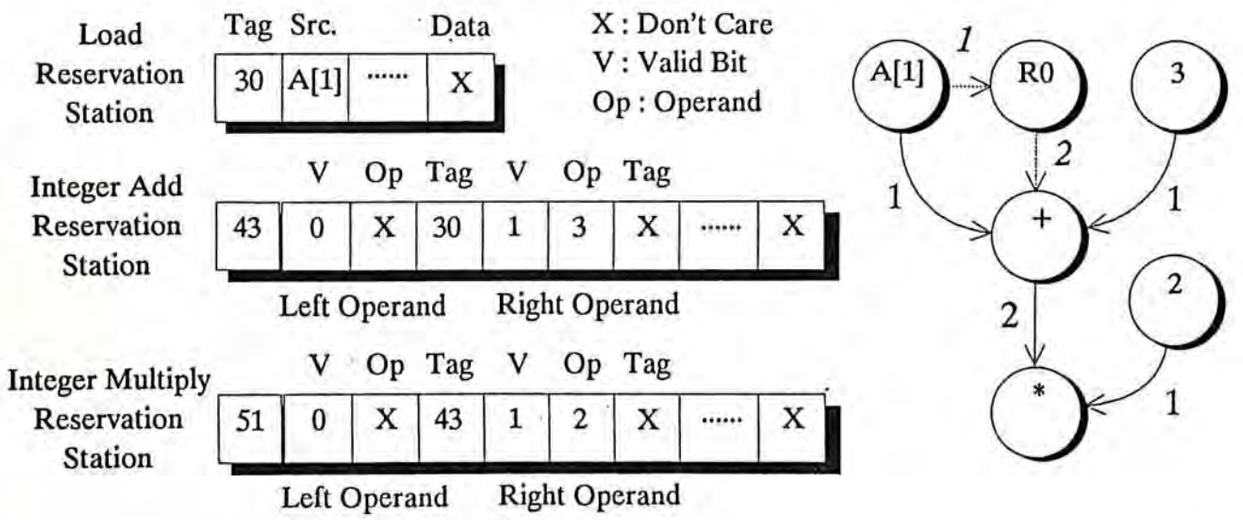
gives rise to the procedure graph on the right (SPs are executed whenever possible. For example, the two dotted transfer arcs are replaced by the arc going from

A[1] to +). The static data dependencies are manifested by the pseudo time labels, which in the Multitag Pool, are represented by the Time fields.

Those transfers labelled by a 1 are free of static dependency and thus can be issued to the respective reservation stations. The Memory Load operation is dispatched to the Load Reservation Station numbered 30 (its tag) while the Add operation is allocated the Add Reservation Station numbered 43. As shown, the dynamic dependency - namely the memory load from A[1] is still left unresolved (a cache miss may lengthen the memory load latency) and the left operand of the add operation is not valid.

Similarly in the next cycle, the transfers now with its Time field equal to 2 are issued. As shown in the Multiply Reservation Station numbered 51, the tag 43 has been recorded in the tag field of its left operand, which implies that there is another dynamic dependency - which arises because of the (add) operation's latency.

Figure 6.4. Resolving dynamic dependencies by using Reservation Stations



The system will then proceed asynchronously and autonomously. As shown, the value 30 is recorded in the tag field corresponding to the left operand. When the content of A[1] is finally available on the bus with the tag 30 upon finishing the load

operation, the add reservation station (with a matching tag) will gate in the data, thus making its left operand valid. The dynamic dependency is then resolved and the add operation is ready to fire. This way, the maximum possible lookahead can be achieved. Besides, we have added an element of machine-independence to the procedure graph optimization approach.

One consequence of using reservation stations is that we have a new definition of an arithmetic node and the firing rule. Originally we say that as soon as all operands of an arithmetic operator are ready, the operation will be fired immediately. But now reservation stations are given more self-control. We can have multiple reservation stations so that more that one pending operation can be kept track. Every time when the arithmetic operator is free, it is arbitrated among the ready-to-go operations.

With reference to figure 6.1, we have decoupled the reservation station from the arithmetic node (the functional unit) where an operation with resource conflict or any other dynamic dependency can wait for the particular event(s) to occur. By storing back the computation results given by the functional units into the corresponding reservation stations, we have added to the S-Prototype an extra capability of 'recall'. This provides an insight into the possible implementation of dynamic (hardware-level) common sub-expression optimization.

A point which merits further consideration is that all reservation stations (but not their respective functional units) are now given names - the tags. The effectiveness of this approach has lead us to evaluate the feasibility of allowing the explicit addressing of hardware by instructions. In the final section, we will come to this issue again.

6.4.5 Extracting Parallelism

The identification of parallelizable operations is a dual problem. First, we need a scheme to encode the precedence constraints among the candidate instructions. Then

optimization rules are applied to move the operations back and forth to arrive at an efficient schedule.

Although hardware dependency resolution schemes have obvious advantages over its software counterpart (for example, the actual memory addresses will not be known until execution time. This inhibits the application of many software optimization techniques at compile time), for years, software parallelization or optimization has dominated the mind of the practitioners. The common consent is that the increase in complexity will unavoidably stretch the cycle time. But with advances in hardware technology, we will see many of the compiler optimizations (especially those which are not achieved satisfactorily) being delegated to the hardware.

Precisely, the S-Prototype extracts parallelism in two ways : expedite the issue of operations (as soon as all static data dependencies are resolved) and to override relay of data transfers using SP transformations.

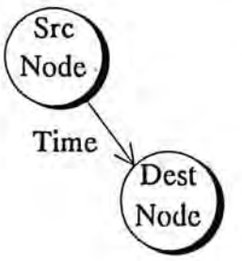
6.4.5.1 Representing Data Dependency in the Multitag Pool

The underlying principle of the S-Prototype is that algorithms will be represented by procedure graphs with causality of the computation encoded by the pseudo time labels. Thus we need a compact scheme to represent a procedure graph at the hardware level. Tags as backward pointers are suitable candidates.

But as revealed from our earlier study in chapter 5, the use of a single tag as in the IBM 360/91 [Tomasulo67] can achieve real-time optimization only. To have predictive power, we should be able to associate more than one tag with a single node/storage location to represent the possible multiple data transfers leading to it. This has led us to the Multitag Pool design.

Intuitively, the Multitag Pool can be perceived as an array of timed tags, content-addressable by using the Time field as the key. An instruction upon decoding will give rise to one or more data transfers, each represented by a timed tag.

The following figure illustrates the general structure of a timed tag. Semantically, it means : the contents of Src Node will be copied to Dest Node at Time. The corresponding procedure graph is shown on the right.



Tag Number	Src Tag1	Src Tag2	Usage Count	LastCopy	Time	Src Node	Dest Node

The inter-relationships of the data transfer arcs in a procedure graph are represented using backward pointers, which manifest themselves as the Tag Numbers in the Multitag Pool. In terms of graph's terminology, for a given data transfer arc corresponding to a tag G, its immediate predecessor will have its Tag Number stored in the Src Tag1 field of G. Data transfers corresponding to the outputs of arithmetic operators present an exceptional case. Should they have two immediate predecessors (two operands) the Src Tag2 field is used as well.

As highlighted earlier, one way to do optimization in the S-Prototype is by eliminating unnecessary data transfers. To achieve this, we have kept for each data transfer/tag a count its immediate successors - the Usage Count. Besides, a data transfer arc corresponding to a tag G which defines the final value of Dest Node will have its LastCopy flag set to 1 (0 otherwise). Any tag with a zero Usage Count and the LastCopy flag reset will be simply discarded in the issue stage since their presence will not affect the final computation result.

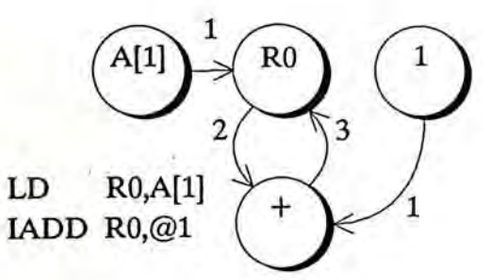
Finally we consider the determination of the Time field. As we have emphasized earlier, the Multitag Pool is only expected to represent the static data dependencies among the data transfers. As a result, the Time field assumes the following definition :

$$\text{Time(G)} = \max\{\ \text{Time(SrcTag1(G))}\ ,\ \text{Time(SrcTag2(G))}\ \} + 1$$

Time(G) denotes the value of the Time field of the tag G while SrcTag1(G) and SrcTag2(G) represents the Src Tag1 and Src Tag2 field of G respectively. Transfer arcs with equal Time field value can be issued in the same cycle. As a special case, a transfer G with the content of its Src Node valid will have SrcTag1(G), SrcTag2(G), Time(SrcTag1(G)) and Time(SrcTag2(G)) all equal to zero. By definition, Time(G) will be set to 1 which implies that the corresponding data transfer can be initiated immediately. The example depicted in figure 6.5 helps to explain the different concepts discussed above.

The definition of Time in fact reflects the rationale behind the S-Prototype design. From another point of view, the Time field helps to establish a partial order for the different data transfers. One can proceed if all transfers with a smaller Time field value have been initiated. There may still be unresolved dynamic dependencies, but we record them with the correct synchronization conditions (e.g. by setting tags) and left the problem behind for the reservation stations.

Figure 6.5. The representation of a procedure graph in the Multitag Pool



Tag Number	Src Tag 1	Src Tag 2	Usage Count	Last-Copy	Time	Src Node	Dest Node
12	0	0	1	0	1	A[1]	R0
13	12	0	1	1	2	R0	IADDL
14	0	0	1	1	1	@1	IADDR
15	13	14	0	1	3	IADD	R0

@1 : A Literal 1
IADDL : Left Operand of an Integer ADD Reservation Station
IADDR : Right Operand of an Integer ADD Reservation Station

As a final comment, there is no denying that the size of the Multitag Pool will affect the overall performance of the system since the fetching and decoding of instructions should be held if there is no free tag. By issuing operations as early as

possible, tags will not be occupied for a long time, and we minimize the risk of running out of free tags.

6.4.5.2 Implementing Transformation Rules

Two transformation rules of the procedure graph theory have been implemented in the S-Prototype, namely the Serial-to-Parallel Transformation (SP) and the Store-Store Cancellation (SSC). The transformations of equivalent graphs are achieved via pointer algebra on timed tags.

In general, any timed tag G with SrcTag1(G)$\neq$0 can be a candidate for an SP transformation. The reader is reminded that SPs are performed at the same time when instructions are decoded. In other words, there will be no intermediate representation.

Algebraically, to perform an SP on tags G_S and G where SrcTag1(G)=G_S, several operations are involved which are summarized below :

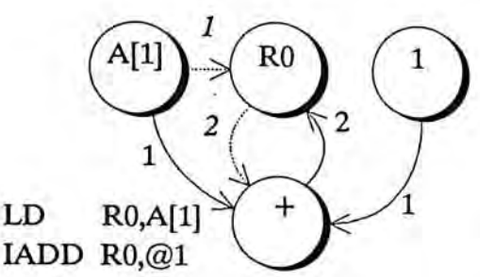
SrcTag1(G) ← SrcTag1(G_S)
SrcTag2(G) ← SrcTag2(G_S)
SrcNode(G) ← SrcNode(G_S)
Time(G) ← Time(G_S)
UsageCount(G_S) ← UsageCount(G_S) - 1

On the other hand, Store-Store Cancellations are not performed via explicit manipulation of timed tags. As revealed in our earlier discussions, during the issue stage, a timed tag with a zero value in the UsageCount and its LastCopy flag reset will be simply discarded since it represents a dummy data transfer. Possibly it is because the data copied to DestNode by this data transfer will soon be overwritten by another arc going into the same DestNode before it can be used.

To conclude our discussions, we come back to our earlier example in section 6.4.5.1. As shown in figure 6.6, an SP has been performed by replacing the two dashed

arcs with the one going from A[1] to + directly. Please note that the transfer arc represented by the tag numbered 12 is now dummy as it will be eliminated by performing the SSC(+R0,A[1]R0). More importantly, the add operation can be fired earlier.

Figure 6.6. Achieving an SP



Tag Number	Src Tag 1	Src Tag 2	Usage Count	Last-Copy	Time	Src Node	Dest Node
12	0	0	0	0	1	A[1]	R0
13	0	0	1	1	1	A[1]	IADDL
14	0	0	1	1	1	@1	IADDR
15	13	14	0	1	2	IADD	R0

@1 : A Literal 1
IADDL : Left Operand of an Integer ADD Reservation Station
IADDR : Right Operand of an Integer ADD Reservation Station

6.4.6 Out-Of-Order Issue and Execution

Having identified operations that can be executed in parallel, we still need a mechanism for out-of-order issue. Instructions held because of unresolved dependencies should not inhibit the issuing (and execution) of succeeding instructions if they are in fact ready to be fired. State it in another way, the processor should best lookahead into the sequential instruction stream as far as possible to search for ready-to-go operations. Only with out-of-order issue the artificial (and often unnecessary) ordering of instructions implied by a sequential programming language can be overridden.

To implement out-of-order issue, some kind of buffering should be adopted between the fetch stage and the issue stage for holding operations/instructions where they wait for their respective static data dependencies to be resolved. The Multitag Pool is dedicated for this purpose.

As revealed in the earlier discussion, operations are scheduled by considering their static data dependencies only. The original artificial ordering of instructions has been broken, but with the necessary causality relationships preserved. Operations corresponding to tags with a 1 in the Time field are ready to be initiated irrespect to their order in the original code sequence, while others are held in the Multitag Pool waiting for their respective static data dependencies to be removed. At the same time, the decoder continues to examine succeeding operations in the dynamic instruction stream and again, tags will be used to represent them. If they are free of any static dependency, their issues can be expedited earlier than those already in the Multitag Pool.

After an operation has been issued to a reservation station, the control can be asynchronous and multi-threaded. With tagging and forwarding, a reservation station is given the greatest autonomy. As soon as all the dynamic dependencies are resolved, the pending operation can be fired. As a result, out-of-order execution and maximal parallelism are achieved naturally.

6.4.7 Memory Accesses

Memory accesses represent a substantial proportion of processing/instructions and they usually have long latencies. As a result, if they are not handled properly, the overall performance will be degraded significantly.

To avoid inconsistencies and to allow for branch repairing (as we have seen in section 6.4.2.2), all memory store operations are totally ordered. By pipelining the store operations, more than one outstanding store operation can exist in each cycle. In the S-Prototype, a maximum of 2 store operations can be in progress by implementing a two-stage pipeline for the MSU. However, writing the same memory location simultaneously

is still prohibited. In other words, we are forced to live with output dependencies. Such kind of memory address disambiguration will be handled by the MSU.

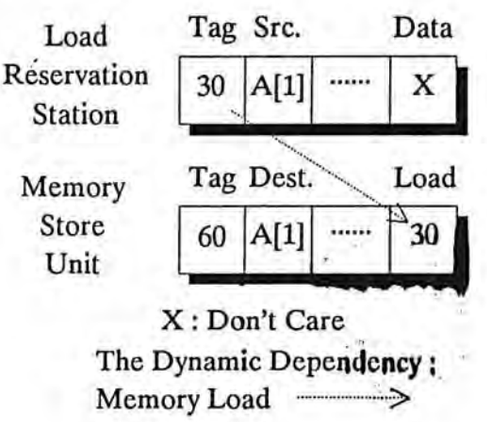
On the other hand, memory load operations give us more autonomy. One more dimension of concurrency is allowed beyond pipelining - multiple load operations can be initiated in the same cycle even if their source addresses are in fact the same. In other words, if we have two individual memory load units each pipelined into 2 stages, then at most there can be 4 individual load operations in progress. Moreover, they are accessing 4 different memory locations since repeated load operations to the same piece of data will be optimized at the issue stage by applying the SP transformation.

However, we still cannot load and store the same memory location at the same time. Because of memory load latency, antidependencies of memory accesses result in loss of performance. To safeguard consistency, each entry in the MSU maintains a Load field. Suppose we have the following code sequence :

```
LD    R1,A[1]
ST    A[1],R2
```

The first load operation has been issued to a load reservation station, say numbered 30 (its tag). Then the MSU entry (numbered 60) allocated for the store operation will have its Load field set to 30. Upon completion of the load operation, the data (the content of A[1]) as well as the tag 30 will appear on the N-bus. The MSU entry with the matching value 30 in its Load field will recognize that its blocking condition has been resolved.

Figure 6.7. Synchronizing Memory Operations By Using Tags and Forwarding



As shown in figure 6.7, the corresponding store operation is now ready for issue. In this way, a synchronization is achieved neatly using tags and forwarding.

6.4.8 Bus Contention and Arbitration

Multiple issues implies multiple completions. This creates the problem of bus contention and arbitration. The Bus Arbitrator shown in figure 6.1 is dedicated for this task. In each cycle, result generated by each functional unit (including load) together with its tag (of the reservation station) will be forwarded to the Bus Arbitrator, where it is maintained in a FIFO queue in the order of its time of arrival. Priority is given to results of loads and operations with long latencies.

With a bandwidth of N for the common data bus, a maximum of N results (with tags) can be broadcasted system-wide in each cycle, which should be the first N elements of the FIFO queue. Results held because of bus contention have to wait for at least one cycle before they can be dispatched.

Chapter 7 An Attempt To Simulate Procedure Graphs Using Graph Grammar

7.1 Introducing Graph Grammar

Graph Grammar refers to a variety of methods for specifying (possibly infinite) sets of graphs or sets of maps. The classical Chomsky String Grammar in formal language theory is useful in describing sets of strings (referred to as string languages). However, when multi-dimensional objects (such as graphs and maps) are involved, the deficiency of string languages becomes apparent. This initiates the study and use of graph grammar in describing structure-furnished phenomena. Significant achievements have been attained in such applications as in pattern recognition, specification of database semantics and biological L-systems.

Casually speaking, as a generalization of string grammar, graph grammar serves as a formal system governing graph rewriting. One starts from a certain graph, then performs successive direct derivations, each time by applying one of the production(s) allowed. In fact, only those derivations suggested by the productions provided are considered to be legitimate. The set of valid graphs obtained define a graph language. In sequential graph grammar, only a certain subgraph is considered and transformed each time. But in parallel graph grammar, repeating sub-structures of the whole graph are transformed simultaneously in each step.

7.2 Basic Concepts in Sequential Graph Grammar

Here we summarize the main points of the algebraic approach to graph grammar. Interested readers are referred to [Ehrig79] and [Ehrig87] for a detailed discussion. As we will be mainly dealing with sequential graph transformations, so let's skip the part on parallel graph grammar. Rigorous mathematical treatments and proofs are avoided, as

our major objective is to give an initial understanding of those terms and concepts which are to be referenced in our succeeding discussions.

7.2.1 Production Rules and Interface Graph

Each graph grammar specifies a set of production rules. To transform a graph G, a certain production is selected. Formally speaking, a production rule P=B1←K→B2 consists of the following three parts :

- The left-hand-side B1,
- The right-hand-side B2, and
- The interface graph K, together with the two graph morphisms b1:K→B1 and b2:K→B2.

The interface graph K consists of the gluing points of the production P. These gluing points can be nodes or arcs, and intuitively, they can be perceived as those parts of B1 (or the occurrence of B1 in the original graph G) which are retained during the direct derivation. Please note that the mappings b1:K→B1 and b2:K→B2 can be non-injective. A mapping F:D→R is non-injective is there exists d1,d2∈D such that d1≠d2 but F(d1)=F(d2).

7.2.2 Gluing Constructions and Pushouts

A graph transformation via a production P is termed as a direct derivation. The process actually involves two pushouts (PO1 and PO2) or gluing constructions (analysis and synthesis), as shown in figure 7.1. In the first step, an occurrence of B1 in the original graph G is located (equivalently the mapping g:B1→G is defined, which can be non-injective also). All but the gluing points of B1 are then removed from G to give rise to the context graph D. With the gluing condition satisfied, B2 is then embedded in D to

produce the daughter graph G'. Corresponding gluing points from B1 and B2 are identified in G', as suggested by their names.

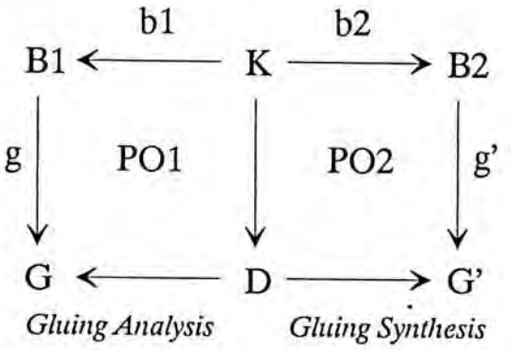


Figure 7.1. A direct derivation consists of two pushouts or gluing constructions

7.2.3 Gluing Conditions

As revealed in the above discussions, whether a production is applicable to a certain graph G is determined by two factors :

- Map a subgraph of G with the left-hand-side of the production, and
- Check if the gluing condition is satisfied by the context graph D constructed.

While the first criteria is self-explanatory, the implication of the gluing condition is worth further consideration. In brief, the gluing condition guarantees that :

- The context graph D fits the basic definition of a legitimate graph, and
- The gluing of D and B1 along K will give back G exactly.

For the first criteria, we have to make sure that all the arcs in D have source and sink nodes. In other words, there exists no dangling arc. Let b1=K→B1 and g=B1→G. The only way a dangling arc 'a' will exist by removing B1 from G is that the source or sink node of 'a' is contained in g(B1), yet it is not a gluing point. In other words, it is not

included in g(b1(K))[1]. Thus by eliminating this possibility, the first condition will be satisfied. Figure 7.2 shows an example. The node "F1" annotated by "a" is the only gluing item. As depicted, the removal of the node labelled "F0" will leave a dangling arc with the node "F2" as the sink but no source node. By definition, D is not a legitimate graph.

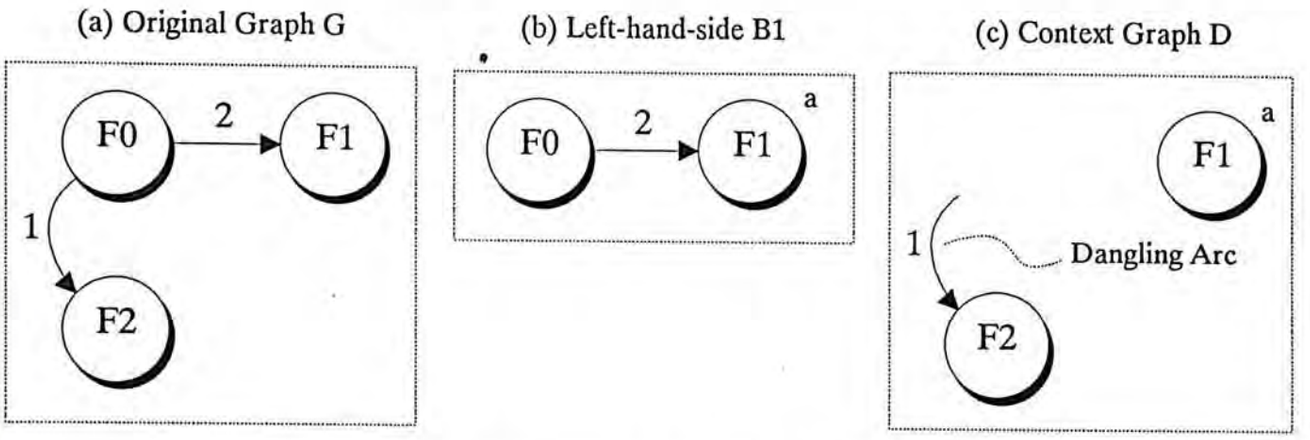


Figure 7.2. Violating the first Gluing Condition - a dangling arc results

The second condition will be fulfilled trivially unless g:B1→G is not injective. That is, some of the items in B1 are identified by g (two different arcs or nodes in B1 are mapped to the same arc or node in G). If it happened that some of these identified items were not gluing points, then although the context graph D could still be constructed, the gluing of D and B1 would not recover G exactly.

Consider the example in figure 7.3. The two nodes annotated by "a" and "b" are the only gluing items. The node annotated by "c" which is non-gluing happens to be identified with "b" in the original graph G (in other words, the mapping B1→G is non-injective). We can see that the gluing of D and B1 will give rise to a graph G' different from the original graph G.

---

[1] Algebracally, we write
$\exists a \in E(G)$ s.t. ((Source(a) $\in$ g(B1) $\wedge$ Source(a) $\notin$ g(b1(K))) $\vee$ (Sink(a) $\in$ g(B1) $\wedge$ Sink(a) $\notin$ g(b1(K)))),
where E(G) is the set of edges/arcs of the original graph G and Source(a) and Sink(a) denotes the source node and sink node of the arc "a" respectively.

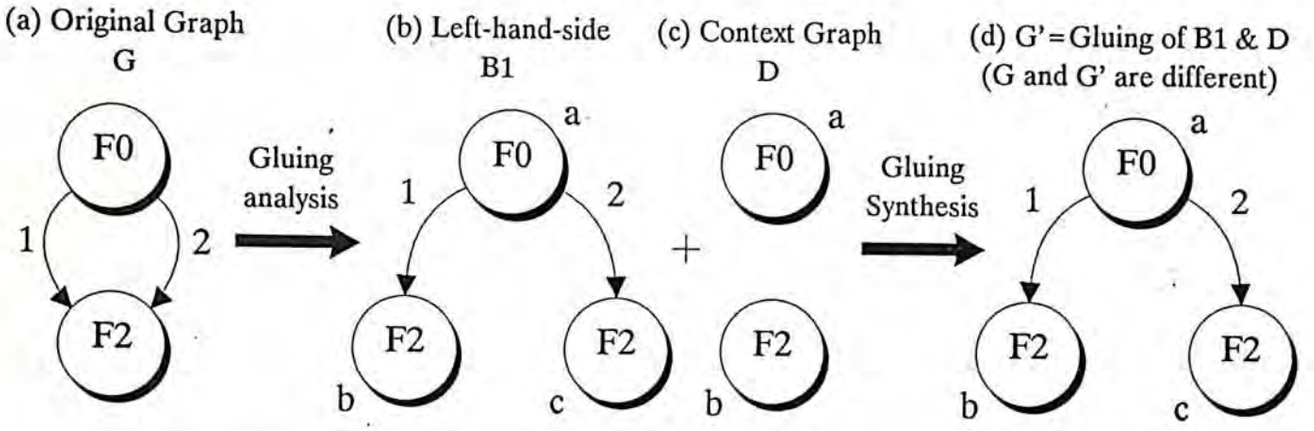


Figure 7.3. An example to illustrate the second Gluing Condition

(Non-gluing items are identified in the original graph G)

7.3 Initial Considerations to Simulate Procedure Graphs

The possibility to specify procedure graphs using graph grammar has been highlighted in [Chen91]. Each computational equivalence identified will give rise to two productions (for the two transformations in opposite directions). With all the vertices in the left-hand-side of each production rule included in the corresponding interface graph K, the gluing condition is satisfied trivially.

Therefore, an occurrence of the left-hand-side B1 in a graph suggests that the corresponding production can be applied. When the economic criteria are favored, the transformation can be carried out. This corresponds to a direct derivation. The right-hand-side is then stitched back to the context graph (obtained by removing the left-hand-side from the original graph) to produce the daughter graph which is equivalent to the original one. The language specified by this grammar will be a finite set of graphs which are all mutually computational equivalent.

7.4 Example

Shown in figure 7.4 is a simulation of a Parallel-to-Serial Transformation by a direct derivation using the production rule B1←K→B2. The nodes "M", "R0" and "R1" are

designated as the gluing points and they are labelled by "a", "b" and "c" respectively. Gluing Condition is satisfied trivially, as

- All nodes in B1 are gluing points. As a result, all the nodes in the original graph G will be preserved during the direct derivation (gluing analysis), thus leaving no dangling arc at all.

- The mapping g:B1→G is injective (in fact, it is a consequence of the semantics of computer operations). Therefore, no two items of B1 (whether gluing or non-gluing) will be identified in G.

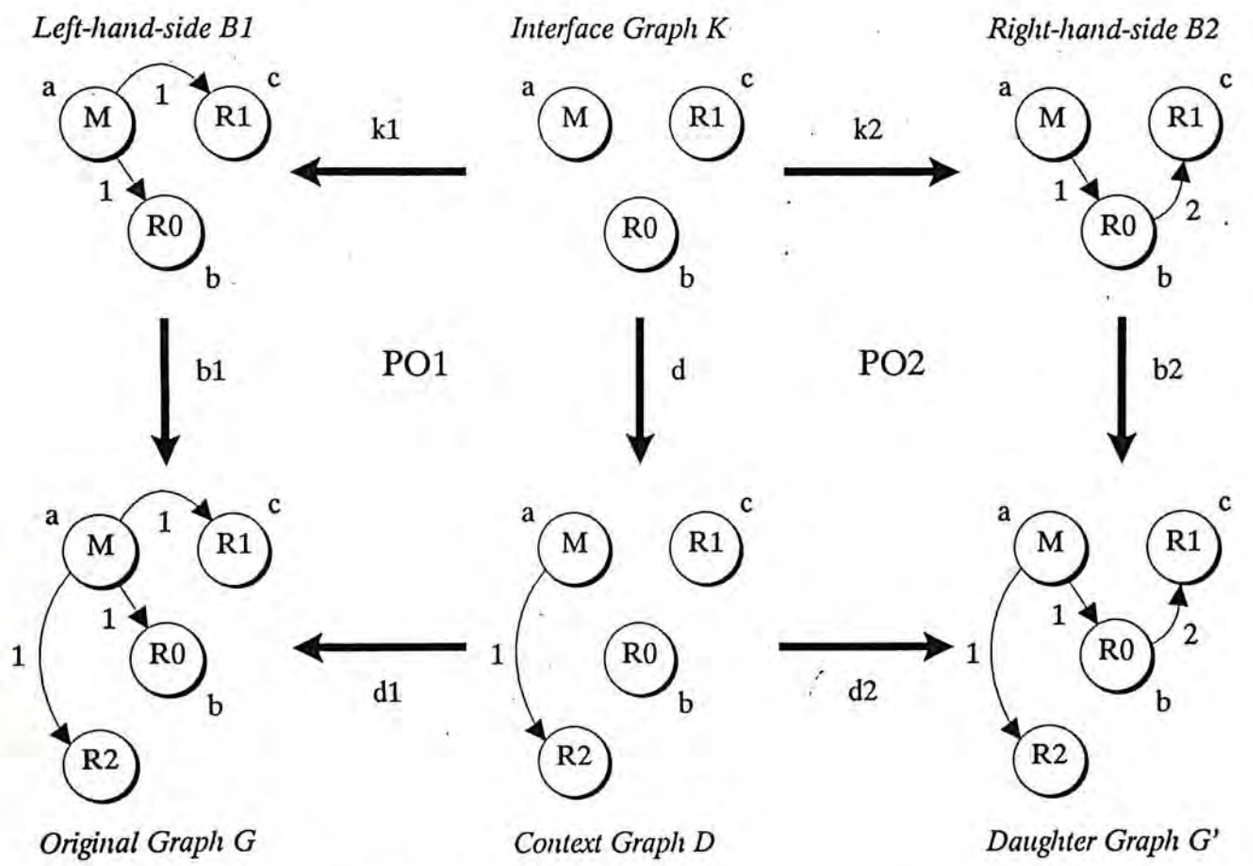
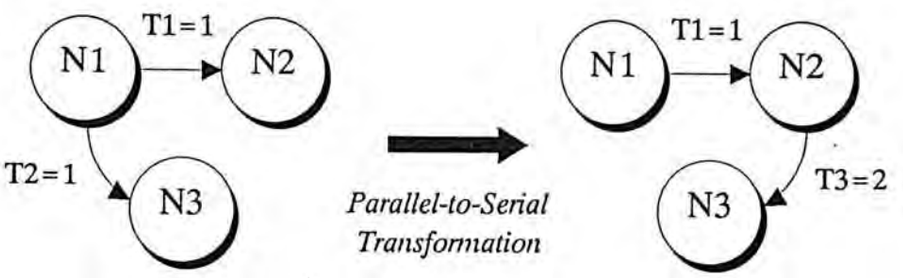


Figure 7.4. Executing a Parallel-to-Serial Transformation via a direct derivation

(k1:K→B1; k2:K→B2; b1:B1→G; b2:B2→G'; d:K→D; d1:D→G; d2:D→G')

7.5 Problems Encountered

However, the actual situation is not that simple. The problem goes with the formulation of production rules. Take an example, suppose the rule of the Parallel-to-Serial Transformation is to be considered, the first attempt may reveal the following :



Doubtless, the result of the transformation is an "instance" of a more general rule only. The same PS transformation can also be executed if both T1 and T2 are equal to 2,3,...,etc. In other words, the colors or labels on the edges of the right-hand-side and left-hand-side are related by some arithmetic operations. In addition, in some cases, the labels may have to exhibit certain arithmetic relationship before a certain production can be applied. There is no denying that it is a violation of the original model of graph grammar in which label alphabets are just sets but not algebras.

Either we have an infinite number of productions, or a new graph grammar model should be invented. As in the above case, we accompany the rule with the equivalence theorem used, and the constraint "T1 should be equal to T2", as well as how the value of T3 can be determined.

But then, direct derivations cannot be done solely on the basis of the structural properties of graphs. Even worse, the various theorems and characteristics of classical graph grammar (such as the notions of parallelism and concurrency) cannot be referred to directly without refinement and justification. Therefore, let's state it clearly - what we

need is a model of graph grammar in which each direct derivation involves only the following :

- Map a subgraph with the left-hand-side of a production rule,
- Check if the gluing condition is satisfied. Hence, all the constraints governing procedure graph transformations should be formulated in terms of the gluing condition, and
- Embed the right-hand-side to produce the equivalent graph.

All other operations (including arithmetics) are considered to be undesirable.

7.6 Some Insights into the Unsolved Problem

Similar problem occurred when one attempted to simulate petri-nets (see [Peterson81]) by graph grammars. Perhaps the analogy described in [Kerowski81] can be drawn. The basic idea is graph restructuring. For a petri-net N with marking M, we construct the related graph G(N.M), in which places and transitions of N are represented by nodes and connections between them by directed edges. The key point is that tokens are no longer considered as labels of places. Instead, they are replaced by additional nodes attached to their places by edges in G(N,M). So for each transition t, assuming that its has r input places (i1,...,ir) and s output places (o1,...,os), we construct the graph grammar production $p(t)=(L(t)\Rightarrow R(t))$ which follows the generic template depicted in figure 7.5.

The darkened nodes attached to each place represent the tokens. By making places and transitions as well as their incident edges as gluing points of p(t), the definition is complete. Thus the firing of a transition t will be equivalent to the direct derivation from G(N,M) to G(N,M') by applying the corresponding production p(t).

More importantly, whether a transition is activated under a certain marking M of a net N can now be determined by examining the gluing condition only. That is, instead of checking each input place of t contains at least one token, we now require that each node corresponding to an input place has at least one darkened node attached to it (a structural property). These input tokens are removed after firing (since they are non-gluing points) and the embedding mechanism will add one token to each output place. To conclude, the relationship that used to exist between labels is now made implicit.

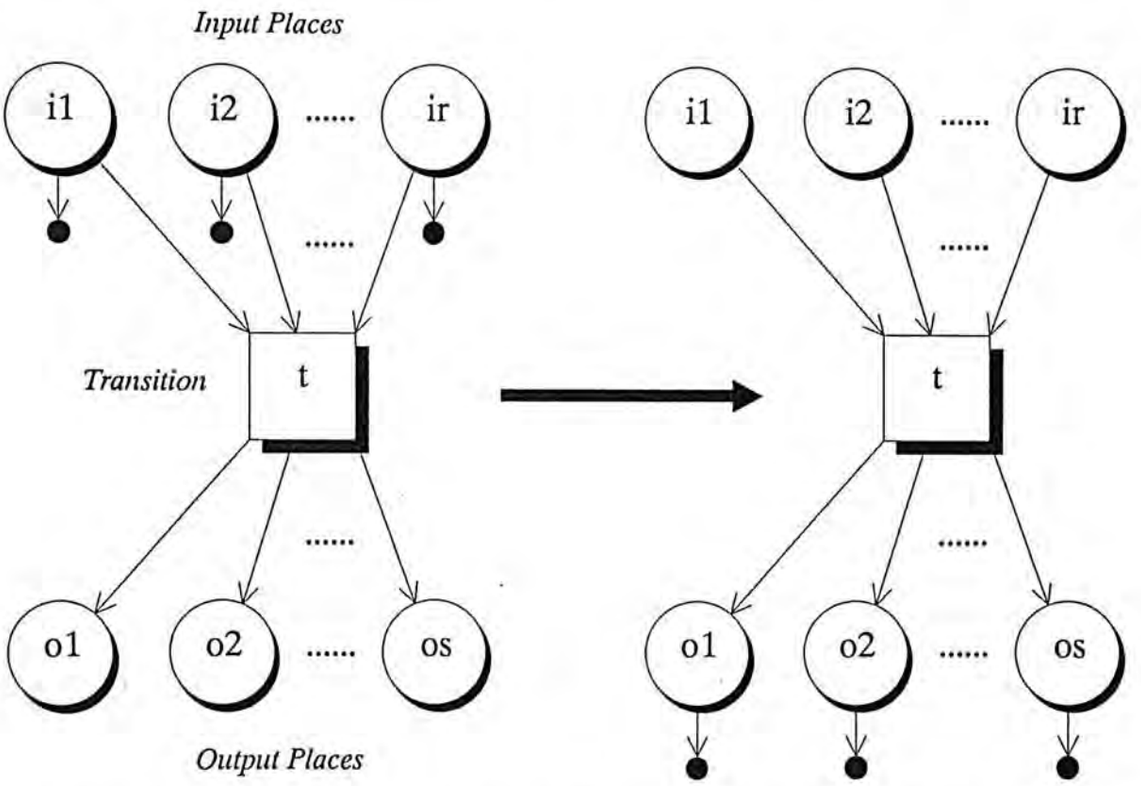


Figure 7.5. The generic pattern of a production rule used for simulating petri-nets

We believe that similar techniques can be applied to our situation by restructuring a procedure graph in some way. Yet, to achieve a true simulation of procedure graphs, other difficulties should not be overlooked. First, in the case of petri-net, we only have to consider whether a token is present in each input place. In other words, it is an existential problem. But the situation now is far more complicated. Prescribed relationship should exist among the magnitudes of the pseudo time labels in order to carry out a certain transformation. It can be expected that such kind of

comparison operations will be difficult to encode using the structure and properties of graphs only. The argument applies also to the many constraints that govern the transformations of procedure graphs.

On the other hand, in the original graph grammar model, to determine whether a production can be applied, we simply need to look for an occurrence of the left-hand-side and check the gluing condition. But for procedure graphs, we sometimes have to confirm the absence of certain (surrounding or background) sub-structure also. For example, consider the situation illustrated in figure 7.6. One necessary condition for establishing the validity of the above Serial-to-Parallel Transformation is that there exists no incoming arc into node N2 with label 2 (see [Chen91]).

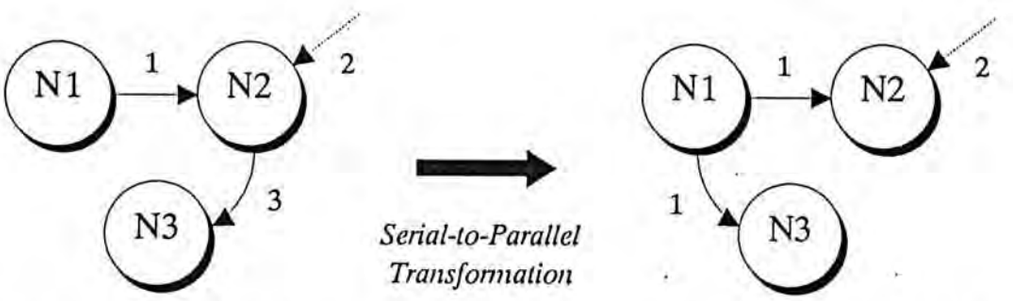


Figure 7.6. To make this SP valid, one should confirm the absence of the dashed arc

Finally, the normalization of pseudo time labels is difficult to achieve using production rules only. In fact, recoloring represents a rather strange kind of graph transformation which has not been well discussed in graph grammar yet. This may call for additional mechanisms other than those provided by the fundamental theories.

To summarize, the original graph grammar model is found to be incomplete for describing the transformations of procedure graphs. Our exploration has at least demonstrated that the procedure graph theory is capable of expressing certain concepts not representable by graph grammar. To supplement its inadequacy, each production rule will be associated with a set of application constraints which should be satisfied before the corresponding procedure graph transformation can be carried out.

7.7 Parallelism, Concurrency and New Transformation Rules

In accordance to the attempt of simulating procedure graphs using graph grammar, transformations via the three classical internal forwarding rules (Serial-to-Parallel Transformation, Parallel-to-Serial Transformation and Store-Store Cancellation) do demonstrate some characteristics of direct derivations, as revealed by the experiments on the Gaussian elimination inner loop (using the simulator discussed in chapter 2). Figure 7.7 depicts the phenomenon of parallelism, in which the order of applications of the transformations SSC(MR,+R) and SP(×R+) is immaterial. In other words, the same final equivalent procedure graph will be obtained.

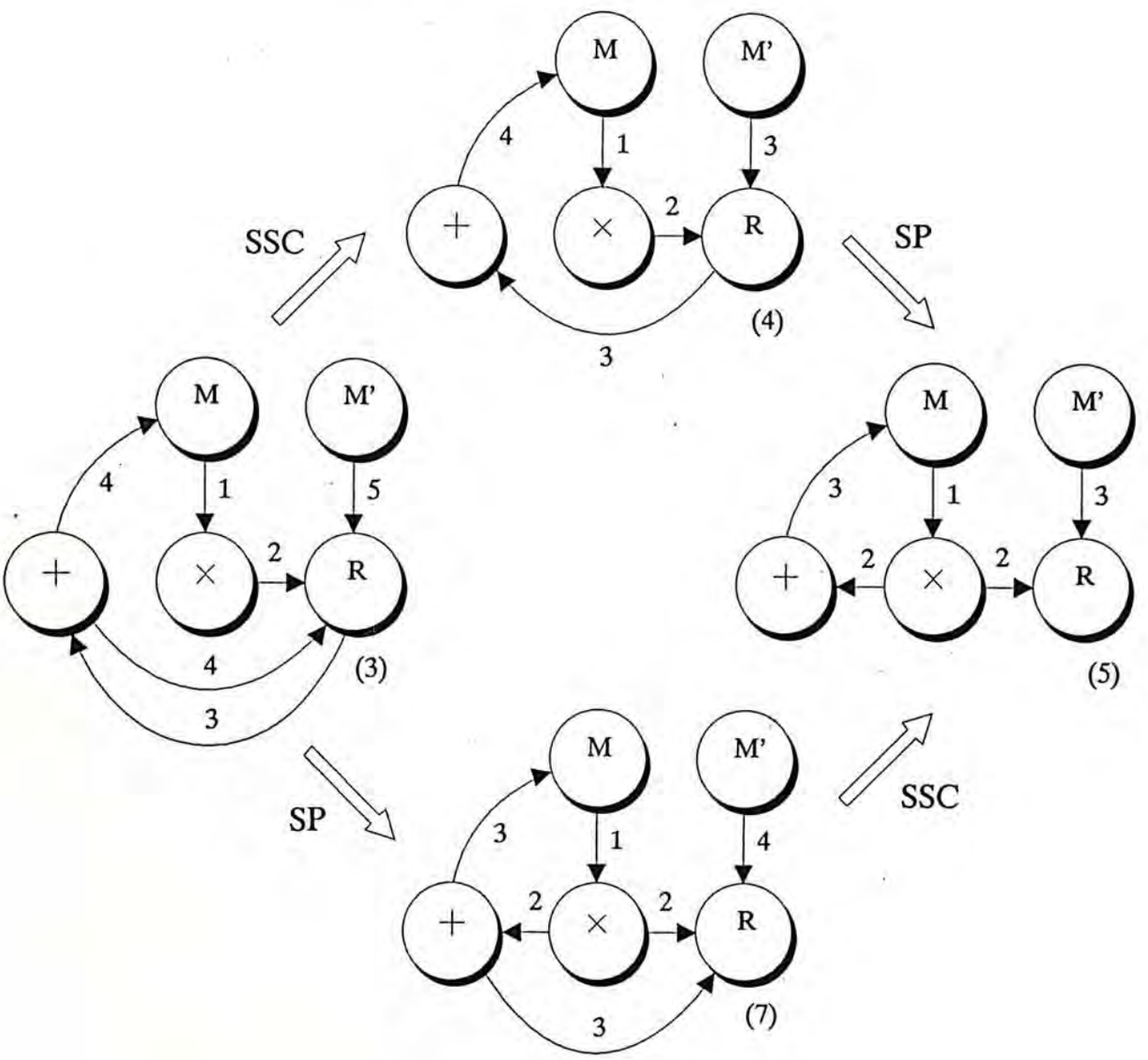


Figure 7.7. An example showing the parallel application of transformation rules
(SSC : M'R, +R; SP : ×R+)

A careful examination should reveal that the two direct derivations or transformations are in fact independent. The "intersections" of the two productions are nodes only which are gluing items. As a result, they will be preserved during the direct derivations. The phenomenon illustrated in figure 7.7 can be explained in terms of the Church-Rosser Theorem of sequential graph grammar [Ehrig83]. Given a sequential independent derivation sequence as in figure 7.8(a), there exists a graph H' which in our case is the procedure graph (7), such that the derivation sequence in figure 7.8(b) is also sequential independent, and that the direct derivations/transformations SSC and SP are parallel independent, as exhibited in figure 7.8(c). Figure 7.8(d) summarizes this Local Church-Rosser Property.

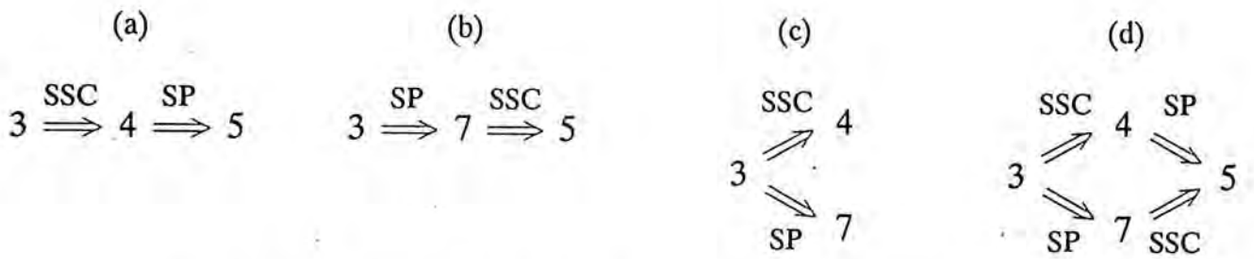


Figure 7.8. The Local Church-Rosser Property as demonstrated in figure 7.7

Moreover, according to the Parallelism Theorem [Ehrig83], there should exist a parallel transformation rule or production "SSC+SP", which upon application, derives the graph (5) from the graph (3) directly (see figure 7.9).

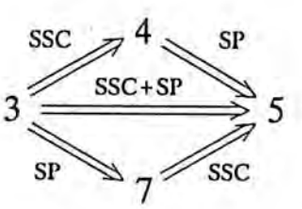


Figure 7.9. SSC+SP

On the other hand, by examining the graph in figure 2.26, one will observe that the transformation sequence SP(ABC) then SSC(DB,AB) occurs frequently (where A, B, C and D are storage nodes). Hence, one may execute the concurrent transformation as shown in figure 7.10. Other examples include 2⇒4, 0⇒2, 3⇒8, etc in figure 2.26.

With reference to figure 7.10, if we associate the transformation SP(+RM) with the production rule SP:B1⇒B2 and the transformation SSC(M'R,+R) with the production rule SSC:B1'⇒B2', we can see that there is some overlapping R between the right-hand-side B2 of SP and the left-hand-side B1' of SSC which is identified in the intermediate procedure graph (8). The fact that some of the items in R are non-gluing (e.g. the data transfer arc from "+" to "R") implies that the derivation or transformation sequence is not sequential independent. In particular, a necessary criteria to establish the validity of the SSC is that the arc from "R" to "M" is removed via the SP. In other words, their order of application does matter.

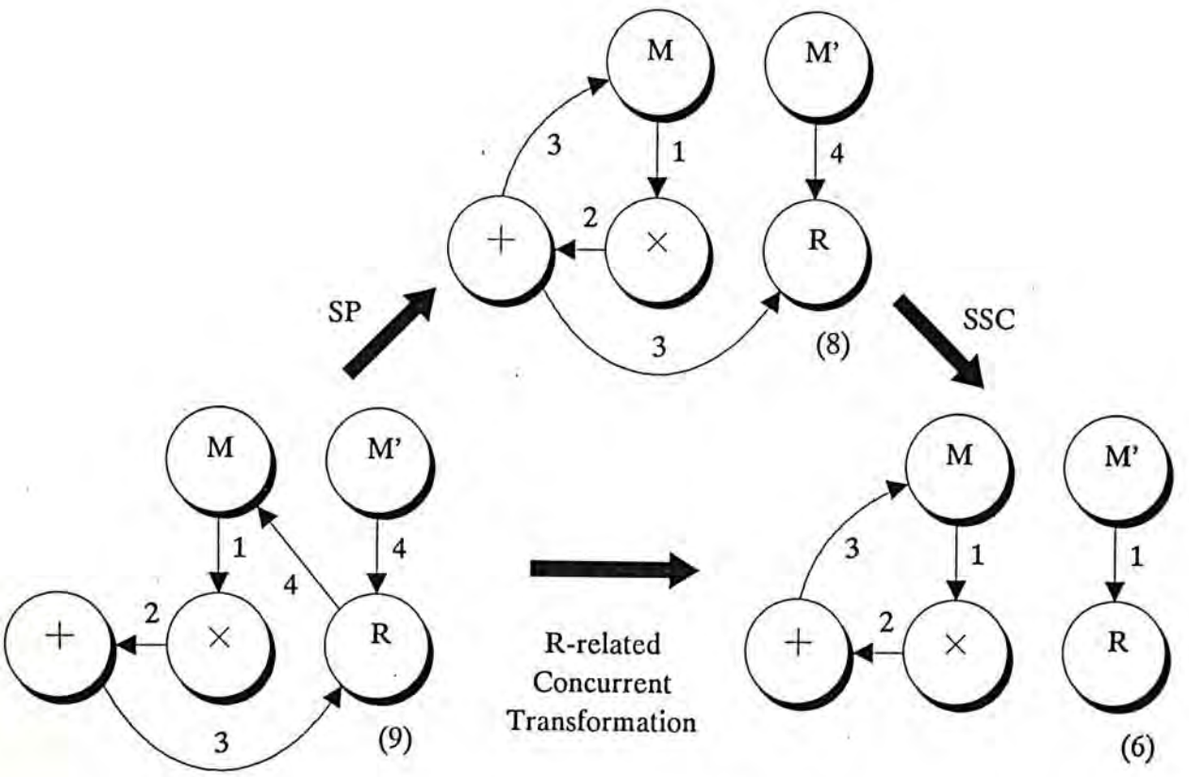


Figure 7.10. The concurrent transformation SP(+RM) followed by SSC(M'R,+R)

In terms of the Concurrency Theorem [Ehrig83], we say that they are R-related and there exists a concurrent production "$SP*_{R}SSC$" which simulates the effects of the two transformations simultaneously. As depicted in figure 7.10, this R-related concurrent transformation rule derives the graph (6) from the graph (9) directly.

Another well-known example is the dL-transformation mentioned by T. C. Chen [Chen91].

Regardless of the Church-Rosser Property, the Parallelism Theorem or the Concurrency Theorem, the direct consequence is that new transformation rules (or productions) are identified. But whether these new "amalgamated" transformations are desirable (or applicable) in actual implementation should be evaluated on the basis of their ease of execution. Just as a final remark, the discussions of the Church-Rosser Property, the Parallelism Theorem and Concurrency Theorem should not be restricted to two productions or transformations only.

Chapter 8 Representing Causality Using Petri Nets

In this chapter, we consider the application of the Petri net Theory as a tool for system modeling. A brief introduction of its characteristics will be given and a Petri net model for program analysis and computer operations optimization will be proposed. Through this study, we hope that we can evaluate the expression power of the procedure graph theory and more importantly, the insufficiencies uncovered, if any, can help to reveal possible extensions to the basic procedure graph theory.

8.1 Defining Petri Nets

To begin, we first introduce some of the concepts of the Petri net theory which will be referred in our succeeding discussions. Basically, we follow the notations defined by J. L. Peterson in [Peterson81], and the original presentations of many of the ideas in this section can be found in [Peterson81] and [Murata89].

A Petri net N can be described as a 4-tuple (P,T,I,O) where P is a finite set of Places ($p_1,...p_n$) and T is a finite set of Transitions ($t_1,...,t_m$). P and T are disjoint and thus $P \cap T = \varnothing$. $I : T \rightarrow P^\infty$ denotes the Input Function while $O : T \rightarrow P^\infty$ is the Output Function which maps each transition t_i to its input places and output places respectively.

P^∞ means the bag of elements from P. A bag is similar to a set but with one exception - duplication of elements is allowed in a bag to cater for the multiplicity of a place (being a multiple input or output place of a transition). As a result, we further define by $\#(p_i,I(t_j))$ the number of occurrences of the place p_i in $I(t_j)$ and $\#(p_i,O(t_j))$ the multiplicity of p_i in $O(t_j)$ respectively.

Tokens reside on places and their distribution is defined by the current marking μ. So, the number of tokens on any place $p_i \in P$ is given by $\mu(p_i)$. Represent by μ_0 the

initial marking of a Petri net before any transition is fired. Tokens are consumed and produced as transitions are fired. The set of different markings which can be obtained through successive firings is denoted by $R(N,\mu_0)$. A place p_i is said to be bounded if

$$\exists M \geq 0 \text{ such that } \forall \mu \in R(N,\mu_0),\ \mu(p_i) \leq M$$

As a final comment, the reachability set $R(N,\mu_0)$ can be infinite if there exists an unbounded place in the Petri net.

8.1.1 Petri Nets as a Tool for System Modeling

The Petri net view of a system concentrates on two primitive concepts : events and conditions. The presence of a certain set of conditions leads to the occurrence of an event which corresponds to the change of state of the system. As a general rule, events are modeled by transitions and places correspond to conditions. When Petri nets are used to model a system, the characteristics of the system under consideration are implemented in two ways :

- *The Structural Properties of Petri nets* : As a general rule, the events of the system are modeled by Transitions and Places are used to represent the conditions. The way places and transitions are inter-connected determine the behavioral properties of the system. In addition, the marking of the Petri net specifies the distribution of tokens on the places. For each place, the presence of tokens as well as the number of tokens present determine whether the corresponding condition represented by the place holds or not. Together, these conditions define the 'current state' of the system.

  As a note, different sub-classes of Petri nets may quote some restrictions on the legitimate structure of a Petri net. For example, In a State Machine (SM), each

transition should have exactly one input place and one output place. Doubtless, these restrictions will in certain degree limit the modeling power of the corresponding sub-class of Petri nets.

- *The Firing rule of Transitions* : The firing of a transition corresponds to the execution of a certain event. At the same time, the generation of a new marking denotes a new state of the system.

  Only enabled transitions can be fired. A transition is said to be enabled when each of its input places is non-empty. But if a place can be a multiple input of a transition (equivalently, we say that the multiplicity of the input place of that transition is greater than one), the required number of tokens should be present in order to enable the transition. Mathematically, we write

  $$\forall p_i \quad \mu(p_i) \geq \#(p_i, I(t_j))$$

  The transition firing rule is somewhat different in the original Petri net theory. In order to enable a transition, we further require that each output place of the transition should be empty, that is

  $$\forall p_i \in O(t_j) \quad \mu(p_i) = 0$$

  As a final comment, the various constraints governing the behavior of the system are specified.

8.1.2 The Characteristics of a Petri Net

As a tool for system modeling, Petri nets possess the following characteristics :

- *Inherent Parallelism, Concurrency and Decision (conflict)* : The order of firing of Independent enabled transitions can be arbitrary. Therefore, Petri nets are

suitable for modeling systems of distributed control with multiple processes executing concurrently in time (an exception being the State Machine subclass of Petri Nets).

- *Asynchronous Nature* : In the original Petri Net theory, there is no inherent measure of time. This expresses a philosophy of time in which the only importance of time is to define a partial ordering for the occurrence of events.

- *Apparent Nondeterminism* : A sequence of transition firings represents the occurrence of a sequence of discrete events. The actual order is one of the possibly many allowed. Yet, the underlying partial ordering is unique.

We can conclude from the above that Petri Net theory is especially suitable for specifying constraints. Analysis of sequential program (instructions) being one of them.

8.1.3 Useful Extensions

To increase the modeling power of a Petri net, several extensions have been made (see [Peterson81]). The introduction of Inhibitors being an important one. So far in our discussion, we have taken for granted the fact that in order to enable a transition t_j, each input place p_i should contain the required number of tokens $\#(p_i,I(t_j))$. But in some circumstances, certain events of the system modeled can only take place if specific condition(s) do(es) not hold. Equivalently, we require that certain input places be empty so as to enable the transition.

While it is possible to test the nullity of a bounded place, the general case when places are allowed to contain arbitrary (or infinite) number of tokens cannot be solved with the concepts mentioned so far. Therefore, a new construct, Inhibitor, is introduced. When an input place p_i is designated as an inhibitor of a transition t_j, the number of tokens on p_i should be zero in order to enable t_j ($\mu(p_i)=0$). With inhibitor, the resulting

extended Petri Net theory can be proved to have the modeling power of a Turing Machine (see [Peterson81]).

Another significant extension is about the concept of time. There is no inherent measure of time in the original Petri net theory and transitions fire in an asynchronous manner. However, when the system modeled is concerned with a 'relative' timing concept only (such as number of working days, number of clock cycles, etc), the inability to measure 'real time' will not be a significant drawback again.

Instead of being represented by a single transition only, each event will now be modeled by a pair of transitions. The firing of the first corresponds to the start of the event while the firing of the second signals its end. Transition firings occur at fixed time intervals. After each fixed time period τ, all enabled transitions are fired, possibly enable other transitions, which will be fired in the next period. For each event, it is the number of time periods N elasped between the firings of its first and second transitions instead of $N\tau$ (which is a measure of the real time) that we are really interested in. As an example, when computer operations are modeled with events correspond to instructions, N gives the number of machine cycles required to execute a certain instruction.

8.2 Program Analysis and Modeling Computer Operations

When a program/algorithm is to be modeled by a Petri net, we are in fact focusing on the flow of control and information as well as the sequencing of the different computations, instead of the actual values computed. When the precedence relationships between computation (or the instructions) are being spelled out, analysis of the program can help to uncover those unnecessary precedence constraints, which when removed, would speed up the execution of the program.

In fact, because of the use of a sequential programming language, the programmer is forced to specify a total ordering for the computations/instructions of the program. However, there is no denying that a partial ordering is enough to express all the necessary precedence constraints between them.

8.2.1 Representing Causality Relationships

Three kinds of precedence constraints govern the sequencing and execution of instructions, namely, Read-After-Write dependency, Write-After-Read dependency and Write-After-Write dependency [Stone87]. To begin with, we first consider their respective representation in Petri net. Assume a three-address code architecture. Instructions can be roughly classified into two categories - Computational Instructions and Control Instructions (e.g. branches). Our focus will be on the former. A computational instruction assumes the following general format :

OP Dest,Src1,Src2

The semantic interpretation is

Dest ← Src1 OP Src2

- Transitions in Petri nets are primitive and their firings are instantaneous. Consequently, it is difficult to express the variable latencies involved in instruction executions. In other words, we need a non-primitive transition as the one adopted in figure 8.1, where a computational instruction is presented by a combination of two primitive transitions and one place.

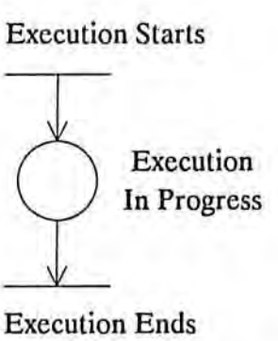


Figure 8.1. An Instruction

The firing of the first transition signals that the execution of the instruction starts while the firing of the second corresponds to the end of its execution. The time delay between the two firings is the latency involved.

An important implication is that parallel execution of independent instructions can now be expressed in detail. Please be noted that the time delay idea can be applied by including as many places as desired between the two transitions if one wants to express that the execution of an instruction takes several stages to complete, with proper time delays as needed.

- Storage locations are manipulated (read or written) by instructions, either as Dest's and/or Src's. Each of them is represented by a pair of places. We are concerned with the possible access conflicts that may exist between instructions. As a result, the reference information about the storage locations (such as data in tags) instead of their actual values are recorded, with one place designated for READ (pRead) and the other for WRITE (pWrite).

- The token(s) on the two places represent the state of the corresponding storage location - namely, its access permit. The rule is simple. More than one simultaneous READs are allowed if there is no outstanding WRITE. However, a single WRITE will suffice to exclude any other READ or WRITE requests.

- On the one hand, the number of tokens on the place pRead represents the number of (executing) instructions currently reading the storage location. Theoretically, this place is unbounded since arbitrary (large) number of instructions can be reading the same location without causing any conflict. On the other hand, the place pWrite should be strictly bounded and no more than one token can reside on it to ensure safety[1].

---

[1] A bounded place p_i is referred as safe if $\forall \mu \in R(N,\mu_0)$, $\mu(p_i) \leq 1$.

- According to the Optimality Conditions specified by R. M. Keller [Keller75], an instruction is ready for execution if

  - there exists no outstanding write to any of its source operands,
  - no executing instruction is currently reading its destination operand, and
  - its destination operand is also free of any outstanding write.

  Otherwise, it should be blocked until the conflict(s) involved has/have been resolved. Equivalently, these three constraints correspond the three types of data dependencies mentioned earlier respectively.

- With inhibitor, Optimality Conditions can be specified and enforced by checking for the absence of tokens in the respective place(s), as depicted in figure 8.2. Mathematically, we write

  - μ(pWrite of Src1) = 0 $\wedge$ μ(pWrite of Src2) = 0
  - μ(pRead of Dest) = 0
  - μ(pWrite of Dest) = 0

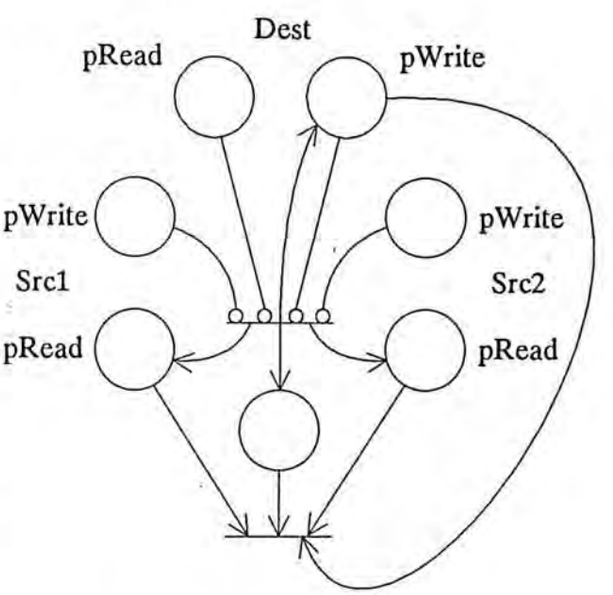


Figure 8.2. Representing the Optimality Conditions [Keller75]

All pRead and pWrite places are empty in μ0. With the access rights of the required storage locations (Dest, Src1, Src2) granted, the first transition is enabled and the execution of the instruction can start. Figure 8.3 shows the respective Petri net representation of the three kinds of data dependencies.

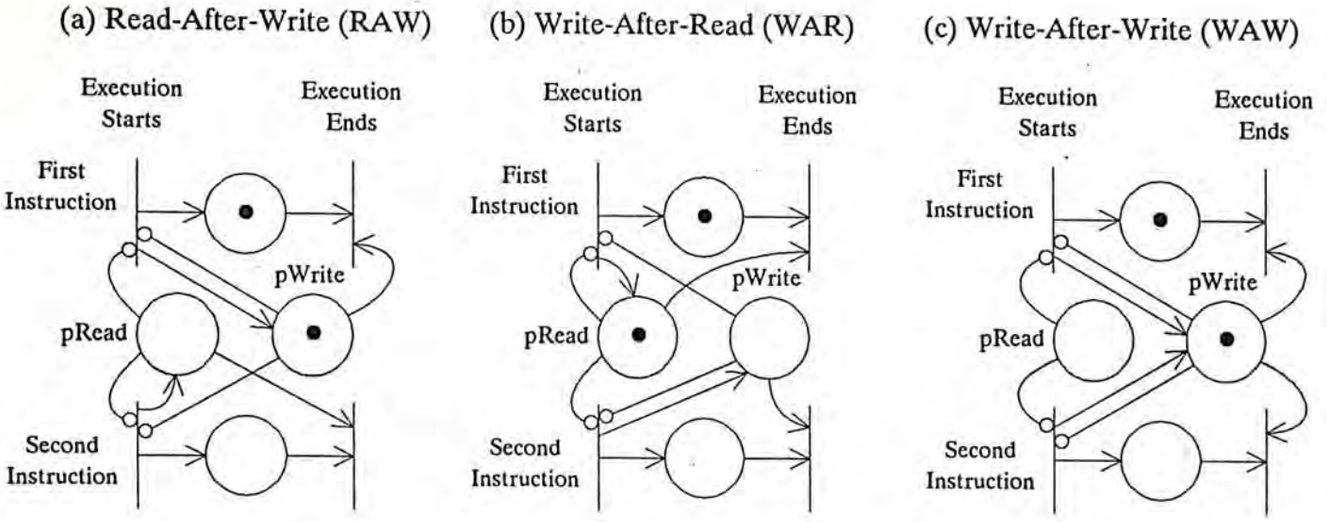


Figure 8.3. Representing the three types of data dependencies [Stone87]

Several points should merit further discussion. First, when the execution of the instruction starts as indicated by the firing of the first transition, a token will be generated in each of its output places, including the pRead place of Src1 and Src2 as well as the pWrite place of Dest. This effectively 'locks' the operands exclusively from other access. The consequence is that the execution of any instruction which has to read Dest and/or write Src1 or Src2 will be held because its first transition cannot be enabled.

The firing of the second transition signals the completion of the instruction's execution. A token is consumed from each of its input places, including the pRead of Src1 and Src2 as well as the pWrite of Dest. Please be noted that these tokens are precisely those generated from the firing of the first transition when the execution started.

To conclude, the executions of instructions are initiated asynchronously when all dependencies are resolved and multiple independent instructions can be executing simultaneously.

8.2.2 Representing the Total Ordering of Instructions in a Sequential Program

When a place is designated as an input place of two different transitions at the same time, a conflict can occur, as exemplified by the two situations in figure 8.4. More appropriate, we say that a "choice" is offered (see [Murata89]). In either case, both transitions t1 and t2 are enabled, and more importantly, they are not independent - the firing of one disables the other. Different orders of firings will give different outcomes.

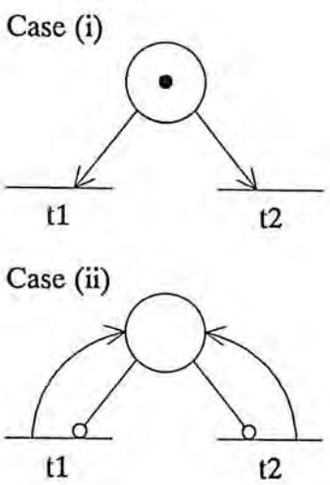


Figure 8.4. Firing conflicts

When applied to instructions, a choice or conflict in transition firings implies an access conflict in storage locations. The different outcomes or firing orders correspond to different execution orders of instructions. Among them, only one is legitimate as dictated by the correct precedence relationship. There is no denying that some kind of 'external constraint' should be applied to resolve the conflict and achieve the desired result. In program analysis, the original sequential ordering of instructions serves this purpose. Consider the following code sequence :

```
ADD   R1,R3,R2
MUL   R4,R1,R2
```

When drawn as a Petri net, the situation is similar to the one depicted in figure 8.5. An access conflict can be identified in R1, namely a Read-After-Write or True Data Dependency. The first transition of each instruction is enabled (let's ignore the dashed

arcs and circle first). This implies that both can start execution. However, the initiation of one will disable the other until its execution completes.

To resolve this conflict, we observe that the original sequential instruction sequence dictates that the ADD should go before the MUL. So we incorporate this constraint into the Petri net by adding the dashed arcs and place. Now, only the first transition of the ADD instruction is enabled and the execution of the MUL will be held. After the ADD finishes, the firing of its second nonprimitive transition will generate a token in the dashed place which allows the execution of the MUL to start. So we have achieve a synchronization. The correct causality relationship is observed to guarantee the consistency of each storage location.

In this Petri net model of program analysis, optimizations are achieved via the removal of unnecessary conditions/orderings (dummy dashed arcs and places), so that independent transitions can be fired simultaneously, corresponding to the parallel executions of independent instructions (total ordering thus becomes partial ordering).

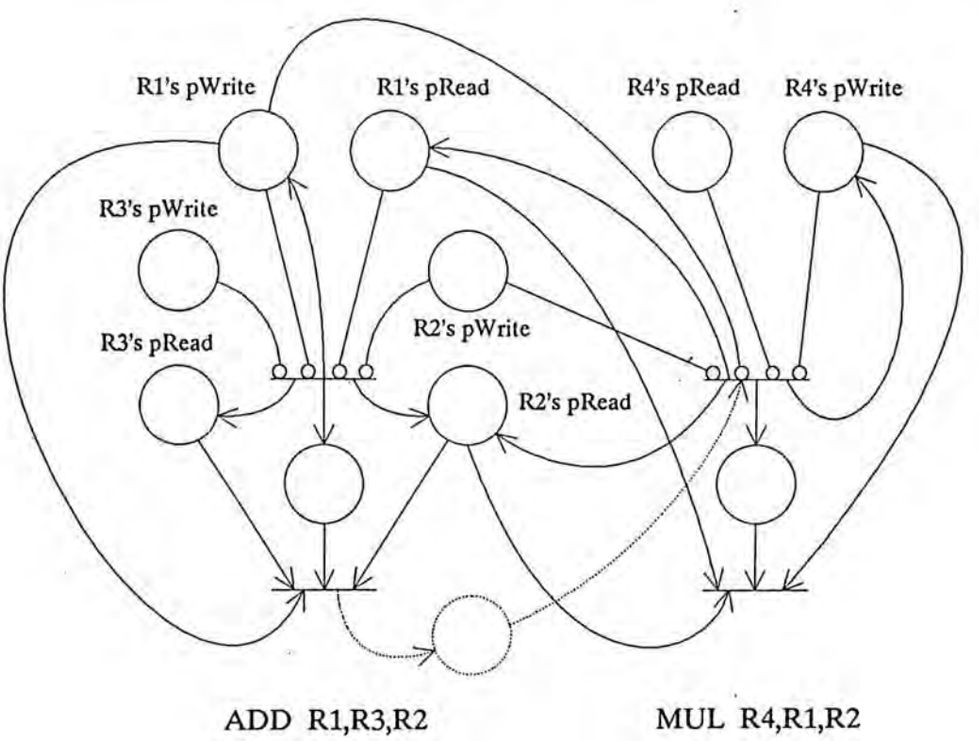


Figure 8.5. Resolving access conflicts using sequential ordering of instructions

8.3 Extending the Model

Simple extensions to the model can greatly increase its modeling power. The sole objective is to obtain a more realistic picture of computer operations.

To begin with, hardware details and the idea of resource allocation can be incorporated. For example, in a machine having two adders, two independent ADD instructions can be dispatched for execution simultaneously. To express this, a place pADD is drawn as in figure 8.6.

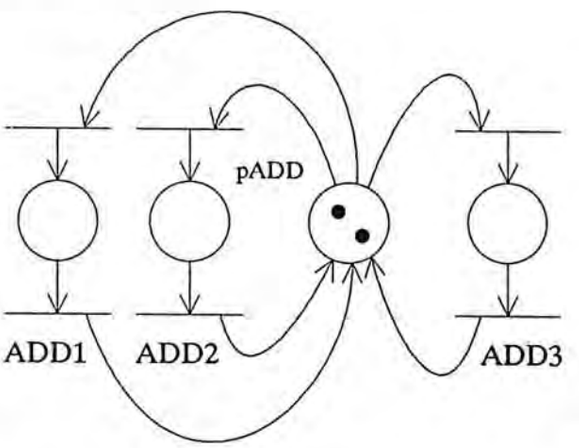


Figure 8.6. 3 ADDs arbitrating for 2 Adders

Initially (in $\mu 0$), there are two tokens in pADD denoting the two free adders of the machine. ADD instructions pending for execution are now subject to another set of constraints other than that presented by data dependence. Those ready for execution have to compete for a free adder (provided that $\mu(\text{pADD}) > 0$) under a certain arbitration scheme. The allocation of a free adder results in the consumption of a token of pADD. If $\mu(\text{pADD}) = 0$, then the first transition of a pending instruction cannot be enabled, effectively holds its execution.

As we have said, the adoption of non-primitive transitions can be further generalized when the execution of an instruction can be partitioned into multiple stages. For example, consider the situation depicted in figure 8.7 where the processing of each instruction goes through four stages.

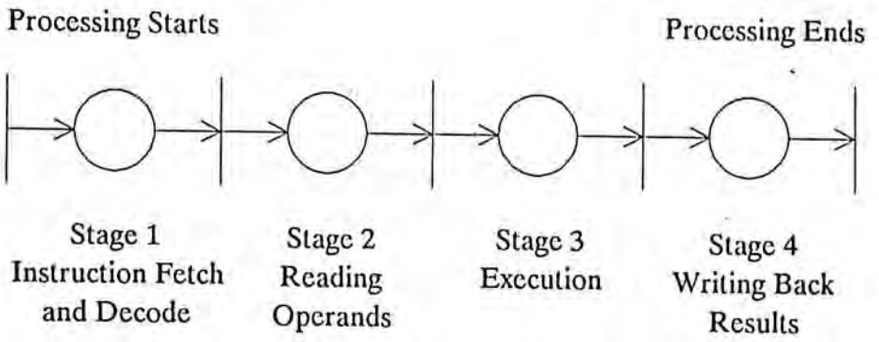


Figure 8.7. The Processing of an instruction described as a 4-stage pipeline

This provides a "microscopic" view of computer operations. Individual stage my have its own specific constraints governing its initiation and no more. Take for an instance, Instruction Fetch should be allowed to proceed as soon as there is a free decoder, even if there is unresolved data dependency, say, a source operand is not ready yet. This should only hold the initiation of the "Reading Operands" stage (and thereafter). From another point of view, a constraint may affect a single stage only. For example, a Write-After-Read dependency exists in the following code sequence :

```
ADD   R1,R2,R3
MUL   R2,R4,R5
```

As the ADD instruction is executing in its "Reading Operands" stage, the MUL instruction cannot proceed with its "Writing Back Results" stage as the corresponding transition is not enabled (see figure 8.8). However, as soon as the ADD finishes reading the source operands, the register R2 is "unlocked" and the MUL can proceed immediately. So, there is no need to wait for the processing of the ADD to complete. The net effect of all these is that maximum overlapping of different operations can now be expressed. For instructions with long latencies, the execution stage can be duplicated to account the multiple cycles required.

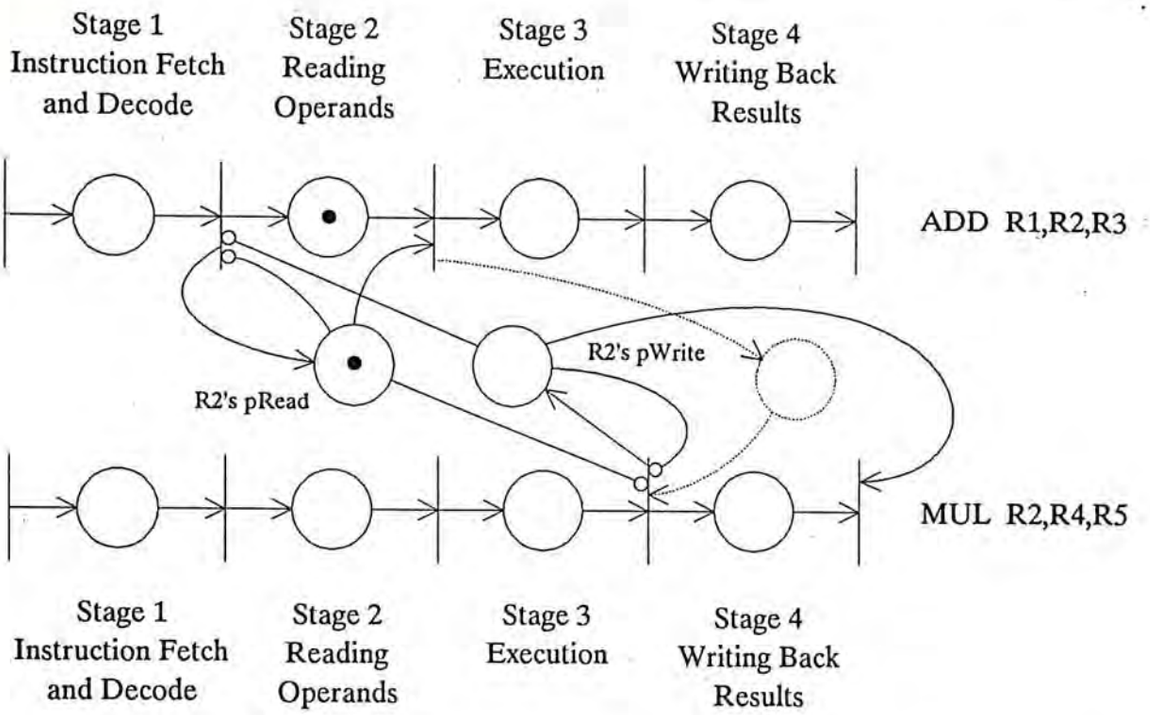


Figure 8.8. Synchronize the 'Reading Operands' stage of ADD with the 'Writing Back Results' stage of MUL

8.4 Comparing Procedure Graphs and Petri Nets

In the last chapter, we concluded that the procedure graph theory is capable of expressing certain concepts not representable using the classical graph grammar model. In this sense, we claimed that the original graph grammar theory is incomplete. Then how about Petri nets?

A careful comparison between the procedure graph theory and the Petri net theory reveals that there is a correspondence between similar constructs and concepts, as summarized by the following table :

	Petri Nets	Procedure Graphs
Conditions	Places and presence of tokens	Pseudo-time labels
Events	Transitions and their firings	Data transfer arcs and firings of operators
Negations	Inhibitors and absence of tokens	T-Operators and F-Operators

Extra information are encoded by procedure graphs. For example, the states of storage locations can be represented using nodes and data transfers are exhibited explicitly by the directed arcs. In addition, such characteristics of Petri nets as inherent parallelism and the capability to express choice or conflict (in occurrences of events) are also possessed by procedure graphs. With the introduction of the T- and F-Operators, an element of nondeterminism has been added to the basic procedure graph, facilitating the description of the dynamic behaviors of systems. We think that this capability can be further enhanced by having pseudo-time labels generated as the outputs of certain privileged operations.

J. L. Peterson in his book [Peterson81] proved that Petri nets have the modeling power of a Turing machine. Based on the above discussion, we believe that procedure graphs should have comparable expressive power to describe/simulate most (if not all) of the concepts in computer science.

Chapter 9 Conclusion and Future Research Directions

As the final chapter of this thesis, we summarize here what we have achieved in the past two years. Certain problems are still left unsolved. Many of them are really interesting and challenging. They may shape future research directions.

9.1 Formalizing the Procedure Graph Theory

In the past two years, much effort has been spent in formalizing the procedure graph theory. In return, we now have a much clearer picture of the semantics of procedure graphs - the definition of pseudo-time labels (especially the algorithmic view and vector time labels), their manipulation and normalization, the application constraints of the three classical graph transformation rules (SP, PS and SSC), etc. And possible extension to vector forwarding has been considered.

The attempt to simulate procedure graph transformations using graph grammar derivations, though turns out to be unsuccessful, does reveal certain interesting characteristics of the procedure graph theory. The notion of parallelism and concurrency in equivalent graph transformations has provided insights of inventing new transformation rules based on the "amalgamation" of basic rules (such as the three classical internal forwarding techniques).

Stemming from the success of procedure graph optimization, we then explore some useful extensions to the basic theory. Two new constructs are proposed - the T-Operator and the F-Operator. At the same time, the Firing Rule of arithmetic nodes/operators is also revised. An element of uncertainty is incorporated as a result.

Originally in the basic procedure graph theory, data transfers represented by labelled arcs are invoked sooner or later unconditionally. With these extensions, we can

control the execution of data transfers at the discretion of the presence or absence of certain conditions. More importantly, the choice can be made dynamically and data-driven. Condition and negation can now be simulated conveniently. We believe that the modeling power of the procedure graph theory will be promoted as a result.

As a final comment, we highlight here the possibility of dynamically generating the pseudo-time labels as the outputs of certain "privileged" operations. The problem should be interesting, we think. In addition, more attention should be drawn to graph transformations involving the T- and F-Operators. Some of the rules may have to be modified as a result. When the T- and F-Operators are used to represent user-programmable conditional constructs, equivalent graph transformations involving them will in some sense correspond to optimizations across basic block boundaries. More effective results can be expected.

9.2 Mathematical Properties Of Procedure Graphs

As we have emphasized from time to time, the use of pseudo-time labels is a remarkable innovation to the classical graph theory. Their existence introduces a new dimension to an ordinary graph - the concept of time, which is different from the structural properties implied by the sets of nodes and arcs. The combinatorial explosion observed and mentioned in chapter 2 should be one of the many interesting mathematical characteristics of procedure graphs only, which should merit more study effort. In particular, every concept and theorem in the classical Graph Theory should have a counterpart in the procedure graph theory, with the added element of time incorporated by the use of the pseudo-time labels. As an example, T. C. Chen has revised the definition of a directed path in [Chen91], giving rise to a new concept - track in space and time.

On the other hand, in accordance to our earlier assertion that a pseudo-time label is in fact the most probable value of a range or a variable (see section 2.3 and [Chen91]), we believe that each pseudo-time label T in any procedure graph can be described as an inequality $T1 \leq T \leq T2$. Given a procedure graph G, the set of all inequalities together dictates the correct causality relationship governing the order of all data transfers and operations in the algorithm manifested by G.

By assigning a value to each of these T's with the corresponding inequality satisfied, another procedure graph G' is obtained such that G' is computational equivalent to G. And by enumerating the set of all combinations of correct values of T's, the set of all equivalent graphs of G can be given rise. We think that such an algebra of inequalities is useful for specifying the "meaning" of a procedure graph as well as proving graphical equivalence.

9.3 Register Abuses[1]

Recall the example mentioned in chapter 2 which concerns the Gaussian elimination inner loop. With reference to figure 9.1, the fact that all data transfers between the arithmetic operators and the memory nodes are channeled through the register R has resulted in a serious performance bottleneck there. Upon successive applications of dL transformations [Chen91], we arrive at the optimized procedure graph exhibited in figure 9.2. The total duration is shortened and more importantly. the bottleneck at the register R has been removed. Our experiments with the T-Architecture and S-Prototype give rise to similar results.

[1] The problem was first identified by T. C. Chen and was also thoroughly discussed by T. C. Chen and K. H. Lee.

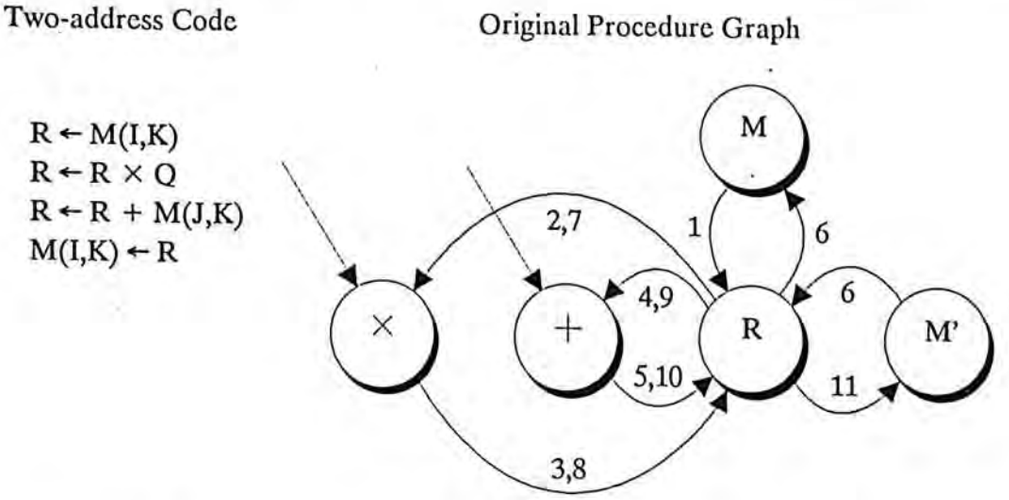


Figure 9.1. Gaussian elimination inner loop (involving 2 elements only)

Recently, RISC architects are proposing the provision of a large number of registers, say 64, 128 or even hundreds. But the example above and our experiments have revealed a very important question - are registers really necessary? What we have observed is that during the processing of the whole array, the register R is updated only once (at the end), although there should be at least one load operation into R in each round of the loop.

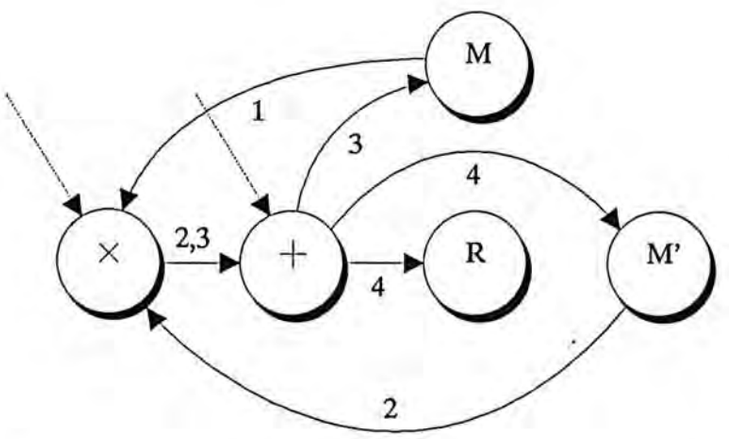


Figure 9.2. The optimized procedure graph for the example shown in figure 9.1

The use of registers as temporary nodes in procedure graphs are highly unfavorable because it will create the following problems :

- There will be a semantics problem since the concept of registers is alien to the higher-level language programmers.

- The use of registers is asymmetric. Various instructions (e.g. data movement instructions) can declare the use of a register but there exists no mechanism to release it explicitly. We just cannot declare that the content of a register is invalid from now on. As a result, the hardware and the compiler have to make a difficult decision before allocating the register to someone else. Moreover, each time during an interrupt, the machine is forced to backup the contents of the whole set of registers. This represents a significant overhead. Perhaps, what we need is an instruction to invalidate the content of a register.

- Data transfers involving registers represent a substantial proportion of the code size. Yet, most of them are dummy. Even worse, relay register transfers (which are frequently created as by-products of compiler optimizations) sacrifice performance much.

- Software optimizations involving registers complicate the design of compilers and lengthen compiling time.

A way out of this difficulty might lie in replacing registers by a hardware stack. Previous results of functional units will be pushed on this stack. References can be served by topping or popping the stack.

9.4 Hardware Representation of Procedure Graphs

Several hardware representation schemes have been considered in our study. Each carries its own advantages and disadvantages. The comparisons done in chapter 6 (see table 6.1 and table 6.2) has favored the use of backward pointers. Based on the design of

the IBM 360/91, the T-Architecture is proposed. Significant enhancements have been incorporated into the original model. Superscalar techniques are applied. With the provision of the Updating Buffer, memory load and store accesses are uniformly handled, effectively facilitating memory data forwarding. On the other hand, with speculative execution, procedure graph transformations can now be applied across basic block boundary. Restricted global optimization as allowed by the maximum branch level is realized at the hardware level. Simulation results reveal that a performance level of 96% of the maximum efficiency can be attained by the T-Architecture. And we have achieved it without the help of software optimizations.

Backward pointers facilitate the implementation of different optimizing strategies such as procedure graph transformations, memory data forwarding and speculative execution. Yet, some problems are still left unsolved.

First, with backward pointers, only upstream arc-information can be conveniently accessed. We have seen how this incapability has constrained the easy application of Parallel-to-Serial Transformations. Perhaps, more global information should be provided. On the other hand, the assumption that only one hardware tag can be associated with each node (storage location) limits us to do real-time optimization only as there is no way to look into events which happen later. Even worse, in case the content of the source is valid already, no forwarding can be done.

In view of these two issues, the S-Prototype is proposed. The Multitag Pool serves a dual function. First, multiple tags can now be associated with a single node which are dynamically allocated from the Multitag Pool (in other words, tags are centralized). In addition, the fact that the Multitag Pool is basically maintained as a doubly-threaded list implies more efficient manipulations of procedure graphs (perhaps saves a search sometimes). To summarize, restricted predictive optimization (although optimization is done ahead of real-time execution, we are still forced to examine the

instruction stream sequentially) as allowed by the availability of tags is realized and we are working towards a hardware implementation of simple compiler optimizing techniques.

9.5 Tags Describing Tags

Procedure graph theory gives rise to a new computation model. A solution algorithm under execution manifests itself as a global procedure graph with pseudo-time labels. An arbitrary subgraph is mapped and extracted which corresponds to a sub-algorithm. An equivalent graph (computation) is identified and the transformation applied. The resulting graph is stitched back into the original graph.

The multiple (timed) tagged representation, being the true image of a procedure graph at the hardware level, realizes this computation model. Please be noted that the requirement that arbitrary sub-graph (or sub-algorithm) can be isolated for consideration is not trivial. We just cannot do this with the simple tagged architecture of the IBM 360/91 or the T-Architecture as we are forced to examine the total computation step-by-step from the very beginning.

Several points are worthy of further discussions. First, as the procedure graph under consideration becomes complicated, the tags will begin to pile up. This creates a representation (and maintenance) problem for the tags. More importantly, as we have mentioned, although optimization can be done ahead of real-time execution with the provision of the Multitag Pool in the S-Prototype, we are still forced to examine the instruction stream sequentially and the manipulation of tags is performed in a similar step-by-step manner. Another mechanism should be provided so that arbitrary set of timed tags can be isolated for consideration, achieving real predictive optimization as suggested in chapter 6. Finally, the issue of how to guarantee the correct simultaneous manipulations of tags has still not been thoroughly considered.

An interesting observation is that the manipulation of (timed) tags, being just another kind of computer operations, should be governed by some precedence constraints also. In other words, while tags are used to represent a procedure graph, there exists another procedure graph describing the correct causality relationship that should be observed by the tags. More importantly, the procedure graph transformations should also be applicable which correspond to the manipulations of tags realizing the optimization of the algorithm under execution.

This "tag-on-tag" (or "meta-tag") view can be further generalized and we believe that its exploration can provide a clearer picture of the exact relationship between the procedure graph and the hardware (e.g., when are transformation rules executed? What timing constraints exist which govern the manipulation of tags?). In particular, the movement of hardware tags as data to process should require datapaths (which may be the standard datapath or a dedicated one). Their specifications can be provided via tags and their manipulations also.

9.6 Software Optimizations

So far in our discussion we have concentrated on hardware optimization strategies. Yet, we have not ruled out the possibility of applying software/compiler techniques. Moreover, we believe that some functions of the decoder can be delegated to a compiler so as to reduce the hardware overhead and shorten the cycle time. A post-compiling scheme similar to the one implemented for the Distributed Instruction Set Computer [Wang&Wu91] can be adopted to extract the underlying data transfers of instructions and to analyze their static data dependencies. The resulting derived object code will then be augmented by precedence information (for example, Time values in the timed tags). Upon decoding, the Multitag Pool can simply base its decision of issue on the pre-determined precedence information.

9.7 Simulation Programs

As mentioned in chapter 2, we have written a program simulating procedure graph transformations. Arbitrary procedure graphs can now be specified conveniently (and interactively) using the graphical interface with its equivalences identified automatically.

On the other hand, with the T-Architecture simulator, we now have a good working environment for evaluating various hardware and software optimization techniques such as the procedure graph transformations and branch strategies. As an example, a preliminary experiment has revealed that by pipelining the functional units, a 30% increase in performance can be obtained for the Gaussian elimination inner loop example.

By iterative re-configuration of the system, we can have a clearer picture of the relationships between the various parameters of the T-Architecture. For example, with a fetch size of 4 instructions per cycle, how many integer add reservation stations are enough? What is the reasonable value of the maximum branch level? With the effects of the various factors on performance correctly defined, we can arrive at a superscalar T-Architecture configuration which is cost-effective and will perform fairly good for a wide range of applications (with varying amount of inherent parallelism).

References

[Acosta et al.86] Acosta, R.D., J. Kjelstrup, and H.C. Torng. "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors." IEEE Trans. Comput., vol.C-35, no.9, pp.815-828, Sept. 1986.

[Chen80] Chen, T.C. "Overlap and pipeline processing." Chapter 9 of Introduction to Computer Architecture. 2nd ed., edited by H.S. Stone, Chicago: Science Research Associates, 1980.

[Chen91] Chen, T.C. "Graphic Equivalence and Computer Optimization." Lecture Notes in Computer Science, Graph-Grammars and their Application to Computer Science, Springer Verlag, Berlin, pp.207-220, 1991.

[Chen&King89] Chen, T.C. and Willis K. King. "Computational invariance and generalized internal forwarding." International Symposium on Computer Architecture and Digital Signal Processing (CA-DSP '89), pp.212-216, Oct. 1989.

[Deo75] Deo, N. "Graph Theory with Applications to Engineering and Computer Science." Prentice-Hall, Englewood Cliffs, N.J., 1975.

[Ehrig79] Ehrig, H. "Introduction to the Algebraic Theory of Graph Grammars (a survey)." Lecture Notes in Computer Science 73, Graph-Grammars and their Application to Computer Science and Biology, Springer-Verlag, Berlin, pp.1-69, 1979.

[Ehrig83] Ehrig, H. "Aspects of Concurrency in Graph Grammars." Lecture Notes in Computer Science 153, Graph-Grammars and their Application to Computer Science, Springer-Verlag, Berlin, pp.58-81, 1983.

[Ehrig87] Ehrig, H. "Tutorial introduction to the algebraic approach of graph grammars." Lecture Notes in Computer Science 291, Graph

Grammars and their Application to Computer Science, Springer-Verlag, Berlin, pp.1-14, 1987.

[Harary69] Harary, F. "Graph Theory." Reading, Mass.: Addison Wesley, 1969.

[Hennessy&Patterson90] Hennessy, J.L. and D.A. Patterson. "Computer Architecture : A Quantitative Approach." Morgan Kaufmann Publishers, Inc., pp.250-349, 1990.

[Hwang&Briggs84] Hwang, K. and F.A. Briggs. "Computer Architecture and Parallel Processing." New York : McGraw-Hill, 1984.

[Hillis&Steele86] Hillis, W.D. and G.L. Steele, Jr. "Data Parallel Algorithms." Communications of the ACM, vol.29, no.12, pp.1170-1183, Dec. 1986.

[Hwu&Patt87] Hwu, W.W. and Y.N. Patt. "Checkpoint Repair for Out-of-order Execution Machines." Proceedings of the 14th Annual International Symposium on Computer Architecture, pp.18-26, June 1987; also IEEE Trans. Comput., vol.C-36, no.12, pp.1496-1514, Dec. 1987.

[Johnson91] Johnson, W.M. "Superscalar Microprocessor Design." Prentice-Hall, Inc., 1991.

[Jouppi89] Jouppi, N.P. "The Non-uniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance." IEEE Trans. Comput., vol.38, no.12, pp.1645-1658, Dec. 1989.

[Jouppi&Wall88] Jouppi, N.P. and D.W. Wall. "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machine." Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp.272-282, April 1989.

[Keller75] Keller, R.M. "Look-Ahead Processors." Computing Surveys, vol.7, no.4, pp.177-195, Dec. 1975.

[Kreowski81] Kreowski, H.-J. "A Comparison Between Petri-nets and Graph Grammars." Lecture Notes in Computer Science 100, Graph Theoretical Concepts in Computer Science, Springer-Verlag, Berlin, pp.306-317, 1981.

[Murakami et al.89] Murakami, K., N. Irie, M. Kuga and S. Tomita. "SIMP (Single Instruction Stream / Multiple instruction Pipelining): A Novel High-Speed Single-Processor Architecture." Proceedings of the 16th Annual International Symposium on Computer Architecture, pp.78- 85, May 1989.

[Murata89] Murata, T. "Petri Nets : Properties, Analysis and Applications." Proceedings of the IEEE, vol.77, no.4, pp.541-580, April, 1989

[Padua&Wolfe86] Padua, D. A. and M. J. Wolfe. "Advanced compiler optimizations for supercomputers." Communications of the ACM, vol.29, no.12, pp.1184-1201, Dec. 1986.

[Peterson81] Peterson, J. L. "Petri net theory and the modeling of systems." Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Pleszkun&Sohi88] Pleszkun, A.R. and G.S. Sohi. "The Performance Potential of Multiple Functional Unit Processors." Proceedings of the 15th Annual International Symposium on Computer Architecture, pp.37-44, May 1988.

[Smith89] Smith, J.E. "Dynamic Instruction Scheduling and the Astronautics ZS-1," IEEE Computer, pp.21-35, July 1989.

[Smith&Pleszkun88] Smith, J.E. and A.R. Pleszkun. "Implementation of Precise Interrupts in Pipelined Processors." Proceedings of the 12th Annual International Symposium on Computer Architecture, pp.36-44, June 1985; also IEEE Trans. Comput., vol.C-37, no.5, pp.562-573, May 1988.

[Smith et al.89] Smith, M.D., S. Lam and M.A. Horowitz. "Limits on Multiple Instruction Issue." Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp.290-302, April 1989.

[Smith et al.90] Smith, M.D., S. Lam and M.A. Horowitz. "Boosting Beyond Static Scheduling in a Superscalar Processor." Proceedings of the 17th International Symposium on Computer Architecture, pp.344-354, May 1990.

[Sohi&Vajapeyam87] Sohi, G.S. and S. Vajapeyam. "Instruction Issue Logic for High-Performance, Interruptible Pipelined Processors." Proceedings of the 14th Annual International Symposium on Computer Architecture, pp.27-34, June 1987.

[Stone87] Stone H. S. "High Performance Computer Architecture." Reading, Mass.: Addison Wesley, pp.102-167, 1987.

[Tomasulo67] Tomasulo, R.M. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units." IBM Journal of Research and Development, vol.11, pp.25-33, Jan. 1967.

[Wang&Wu91] Wang, L. and C.-l. Wu. "Distributed Instruction Set Computer Architecture." IEEE Trans. Comput., vol.40, no.8, pp.915-934, Aug. 1991.

[Weiss&Smith84] Weiss, S. and S.E. Smith. "Instruction Issue Logic for Pipelined Supercomputers." Proceedings of the 11th Annual International Symposium on Computer Architecture, pp.110-118, June 1984; also IEEE Trans. Comput., vol.C-33, no.11, pp.1013-1022, Nov. 1984.

[Wichmann&Hill87] Wichmann, B.A. and I.D. Hill. "Building a Random Number Generator." Byte Magazine, March 1987, pp.127.